

Computer Architecture

Lecture 28: VLIW and Systolic Array Architectures

Prof. Onur Mutlu

ETH Zürich

Fall 2022

10 January 2023

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)

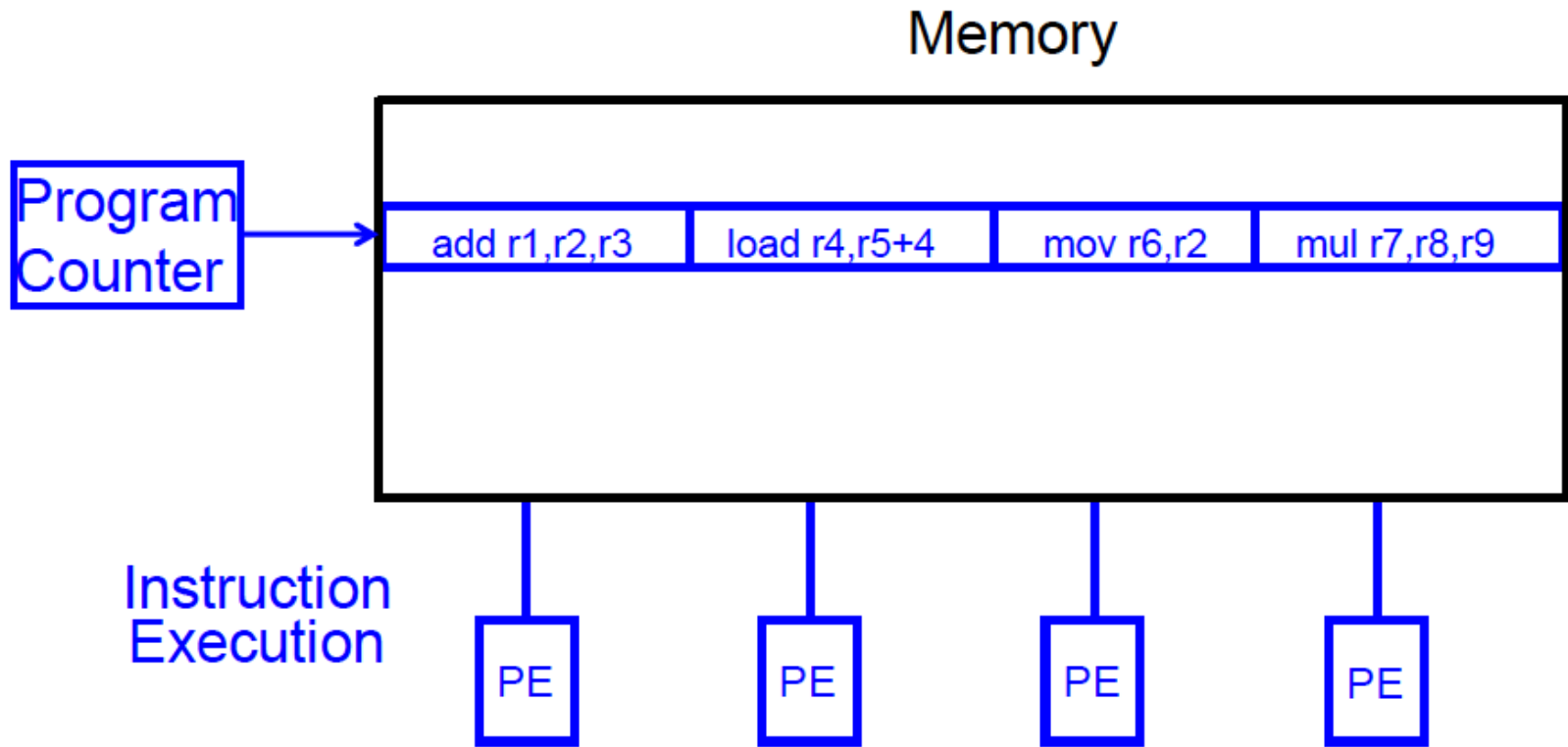
VLIW Architectures

(Very Long Instruction Word)

VLIW Concept

- Superscalar
 - **Hardware** fetches multiple instructions and checks dependencies between them
- VLIW (Very Long Instruction Word)
 - **Software (compiler) packs independent instructions** in a larger “instruction bundle” to be fetched and executed concurrently
 - Hardware fetches and executes the instructions in the bundle concurrently
- **No need for hardware dependency checking** between concurrently-fetched instructions in the VLIW model

VLIW Concept



- Fisher, “**Very Long Instruction Word architectures and the ELI-512,**” ISCA 1983.
 - ELI: Enormously longword instructions (512 bits)

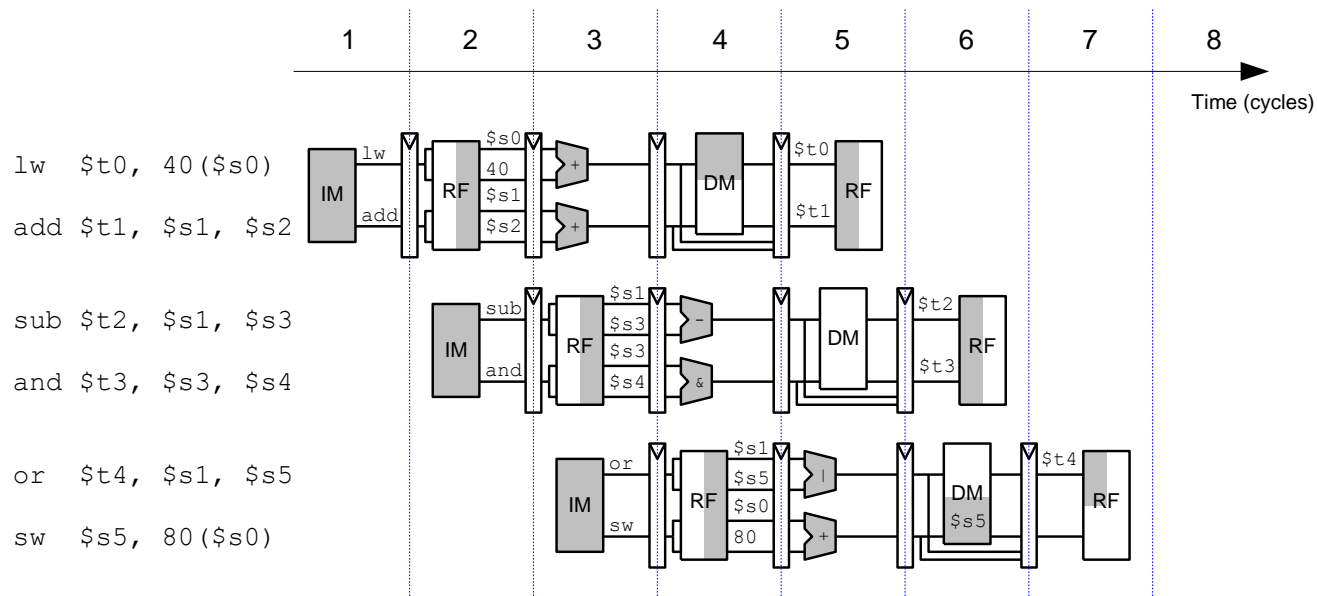
VLIW (Very Long Instruction Word)

- A very long instruction word consists of **multiple independent instructions packed together by the compiler**
 - Packed instructions can be logically unrelated (contrast with SIMD/vector processors, which we will see soon)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional VLIW Characteristics
 - Multiple instruction fetch/execute, multiple functional units
 - All instructions in a bundle are executed in lock step
 - Instructions in a bundle statically aligned to be directly fed into the functional units

VLIW Performance Example (2-wide bundles)

```
lw  $t0, 40($s0)    add $t1, $s1, $s2
sub $t2, $s1, $s3    and $t3, $s3, $s4
or  $t4, $s1, $s5    sw  $s5, 80($s0)
```

Ideal IPC = 2



Actual IPC = 2 (6 instructions issued in 3 cycles)

VLIW Lock-Step Execution

- Lock-step (all or none) execution
 - If any operation in a VLIW instruction stalls, all concurrent operations stall
- In a **truly VLIW machine**:
 - the compiler handles all dependency-related stalls
 - hardware does **not** perform dependency checking
 - What about variable latency operations? Memory stalls?

VLIW Philosophy & Principles

Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
SIGPLAN Notices Vol. 19, No. 6, June 1984

Parallel Processing: A Smart Compiler and a Dumb Machine

**Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau**

**Department of Computer Science, Yale University
New Haven, CT 06520**

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

VLIW Philosophy & Principles

- Philosophy similar to RISC (simple instructions and hardware)
 - Except multiple instructions in parallel
- RISC (John Cocke+, 1970s, IBM 801 minicomputer)
 - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
 - And, to reorder simple instructions for high performance
 - Hardware does little translation/decoding → very simple
- VLIW (Josh Fisher, ISCA 1983)
 - Compiler does the hard work to find instruction level parallelism
 - Hardware stays as simple and streamlined as possible
 - Executes each instruction in a bundle in lock step
 - Simple → higher frequency, easier to design

VLIW Philosophy and Properties

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors) and some ATI/AMD GPUs
 - Most successful commercially
- Intel IA-64
 - Not fully VLIW, but based on VLIW principles
 - EPIC (Explicitly Parallel Instruction Computing)
 - Instruction bundles can have dependent instructions
 - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

■ Advantages

- + No need for dynamic scheduling hardware → simple hardware
- + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
- + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

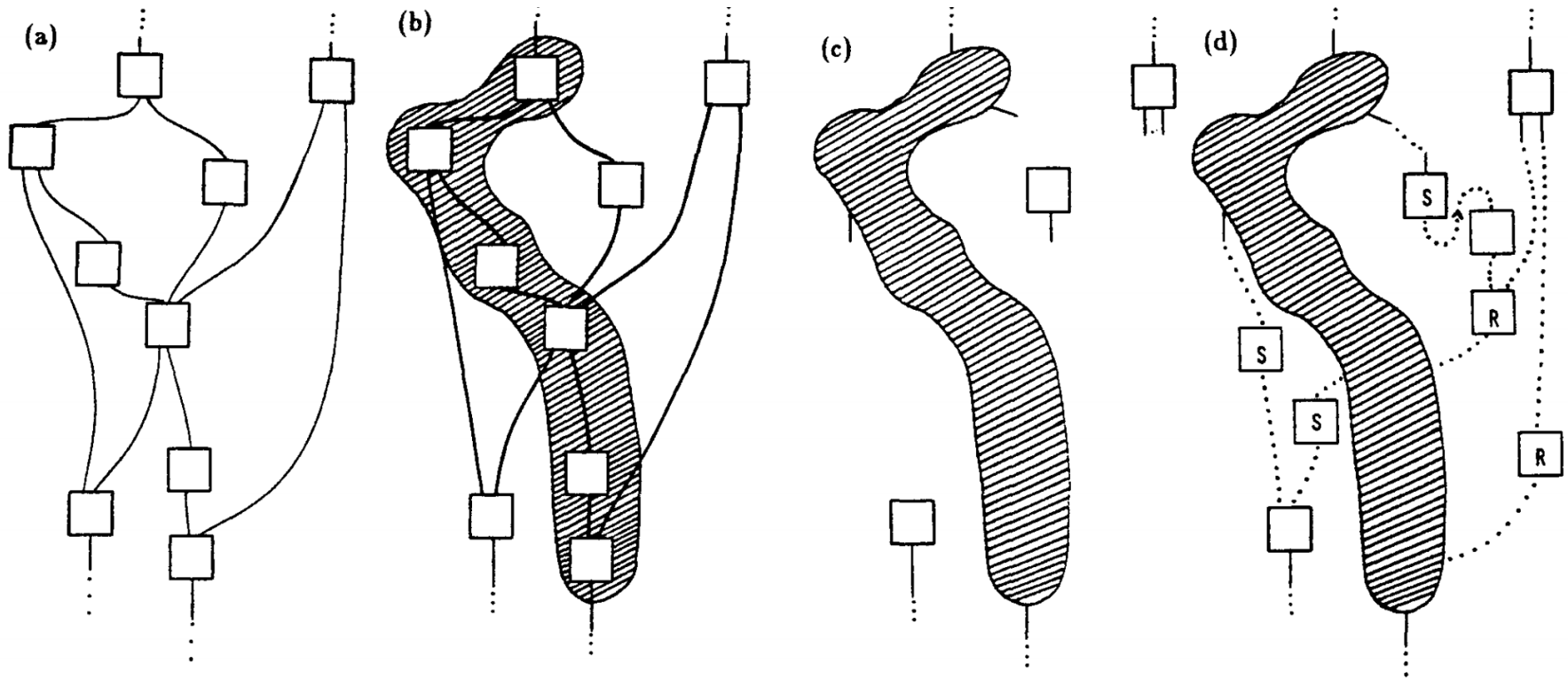
■ Disadvantages

- Compiler needs to find N independent operations per cycle
 - If it cannot, inserts NOPs in a VLIW instruction
 - Parallelism loss AND code size increase
- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- Lockstep execution causes independent operations to stall
 - No instruction can progress until the longest-latency instruction completes

VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
 - Too many NOPs (not enough parallelism discovered)
 - Static schedule intimately tied to microarchitecture
 - Code optimized for one generation performs poorly for next
 - No tolerance for variable or long-latency operations (lock step)
- ++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
 - Enable code optimizations
- ++ VLIW very successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs, GPUs)

Example Work: Trace Scheduling



TRACE SCHEDULING LOOP-FREE CODE

(a) A flow graph, with each block representing a basic block of code. (b) A trace picked from the flow graph. (c) The trace has been scheduled but it hasn't been relinked to the rest of the code. (d) The sections of unscheduled code that allow re-linking.

Recommended Paper

VERY LONG INSTRUCTION WORD ARCHITECTURES AND THE ELI-512

JOSEPH A. FISHER
YALE UNIVERSITY
NEW HAVEN, CONNECTICUT 06520

ABSTRACT

By compiling ordinary scientific applications programs with a radical technique called trace scheduling, we are generating code for a parallel machine that will run these programs faster than an equivalent sequential machine — we expect 10 to 30 times faster.

Trace scheduling generates code for machines called Very Long Instruction Word architectures. In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream. VLIWs are more parallel extensions of several current architectures.

These current architectures have never cracked a fundamental barrier. The speedup they get from parallelism is never more than a factor of 2 to 3. Not that we couldn't build more parallel machines of this type; but until trace scheduling we didn't know how to generate code for them. Trace scheduling finds sufficient parallelism in ordinary code to justify thinking about a highly parallel VLIW.

At Yale we are actually building one. Our machine, the ELI-512, has a horizontal instruction word of over 500 bits and

are presented in this paper. How do we put enough tests in each cycle without making the machine too big? How do we put enough memory references in each cycle without making the machine too slow?

WHAT IS A VLIW?

Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.

More formally, VLIW architectures have the following properties:

- There is one central control unit issuing a single long instruction per cycle.

- Each long instruction consists of many tightly coupled independent operations.

- Each operation requires a small, statically predictable number of cycles to execute.

- Operations can be pipelined. These properties distinguish

The Bulldog VLIW Compiler

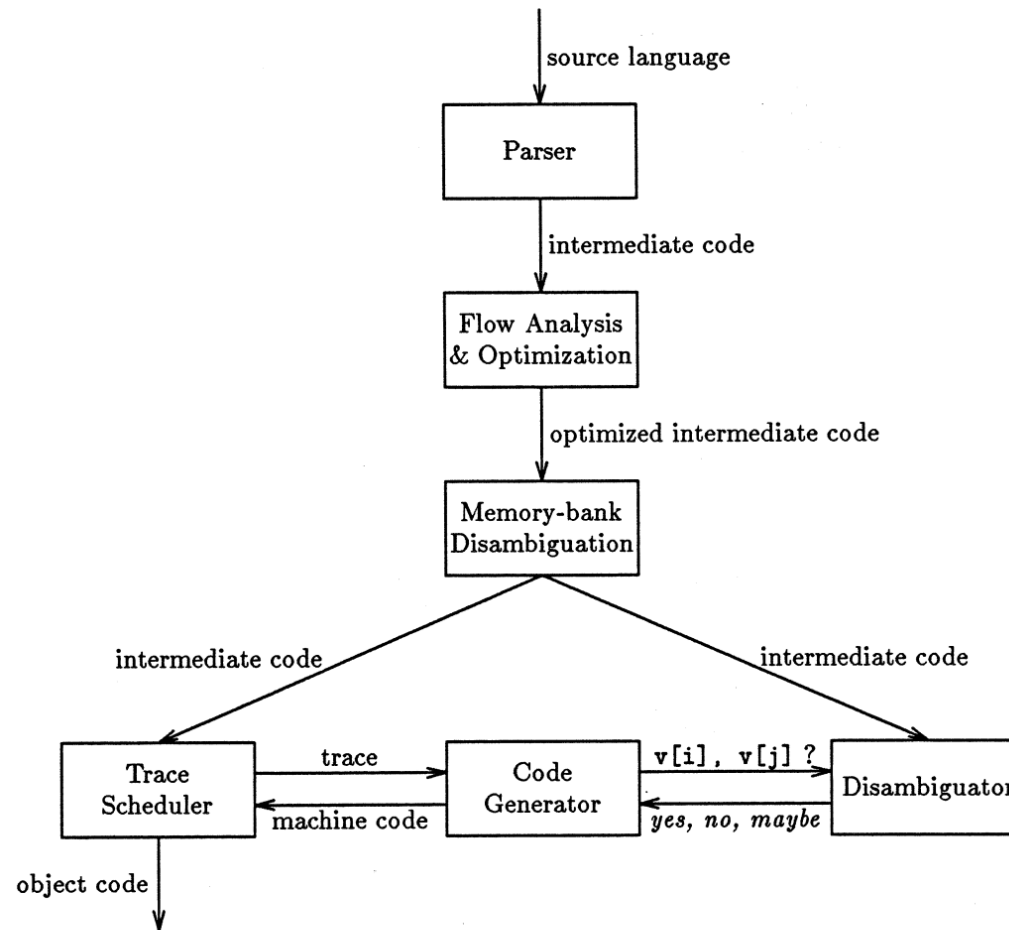


Figure 1.5. The Bulldog compiler.

Another Example Work: Superblock

The Superblock: An Effective Technique for VLIW and Superscalar Compilation

Wen-mei W. Hwu Scott A. Mahlke William Y. Chen Pohua P. Chang

Nancy J. Warter Roger A. Bringmann Roland G. Ouellette Richard E. Hank

Tokuzo Kiyohara Grant E. Haab John G. Holm Daniel M. Lavery *

Hwu et al., [The superblock: An effective technique for VLIW and superscalar compilation.](#)
The Journal of Supercomputing, 1993.

- Lecture Video on Static Instruction Scheduling
 - <https://www.youtube.com/watch?v=isBEVkIjgGA>

Another Example Work: IMPACT

IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

Pohua P. Chang

Scott A. Mahlke

William Y. Chen

Nancy J. Warter

Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

The performance of multiple-instruction-issue processors can be severely limited by the compiler's ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.

Another Example Work: Hyperblock

Effective Compiler Support for Predicated Execution Using the Hyperblock

Scott A. Mahlke David C. Lin* William Y. Chen Richard E. Hank Roger A. Bringmann

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

- Lecture Video on Static Instruction Scheduling
 - <https://www.youtube.com/watch?v=isBEVkIjgGA>

Lecture on Static Instruction Scheduling

Trace Scheduling Idea

(a) (b) (c) (d)

TRACE SCHEDULING LOOP-FREE CODE

43

Lecture 16. Static Instruction Scheduling - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

7,136 views • Feb 26, 2015

46 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
23K subscribers

SUBSCRIBED



Lecture 16: Static Instruction Scheduling
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: Feb 23rd, 2015

Lecture 16 slides (pdf): <http://www.ece.cmu.edu/~ece447/s15/li...>

<https://www.youtube.com/onurmutlulectures>

Lectures on Static Instruction Scheduling

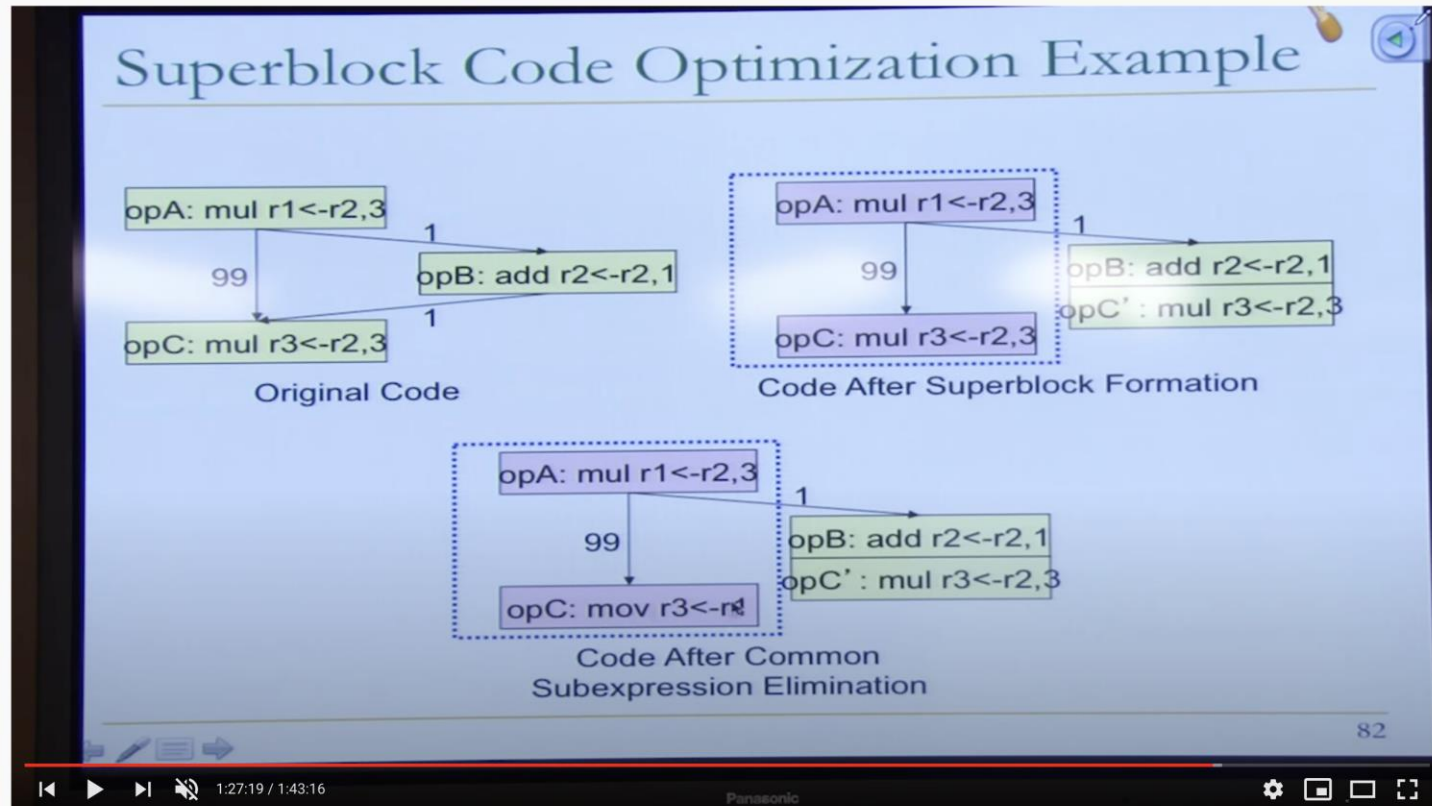
- Computer Architecture, Spring 2015, Lecture 16

- Static Instruction Scheduling (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=isBEVkJjgGA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=18>

- Computer Architecture, Spring 2013, Lecture 21

- Static Instruction Scheduling (CMU, Spring 2013)
- <https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21>

A More Compact Version...



18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5

4,696 views • Sep 23, 2015

41 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
23K subscribers

Lecture 5: Advanced Branch Prediction

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 16, 2014.

Lecture 5 slides (pdf): <http://www.ece.cmu.edu/~ece740/f15/li...>

Lecture 5 slides (ppt): <http://www.ece.cmu.edu/~ece740/f15/li...>

<https://www.youtube.com/onurmutlulectures>

A More Compact Version...

- Computer Architecture, Spring 2015, Lecture 5
 - Advanced Branch Prediction (CMU, Spring 2015)
 - <https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4>

Aside: ISA Translation

- One can translate from one ISA to another *internal-ISA* to get to a better tradeoff space
 - Programmer-visible ISA (virtual ISA) → Implementation ISA
 - Complex instructions (CISC) → Simple instructions (RISC)
 - Scalar ISA → VLIW ISA
- Examples
 - Intel's and AMD's x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
 - Transmeta's x86 implementations translated x86 instructions into "secret" VLIW instructions in software (code morphing software)
- Think about the tradeoffs

Transmeta: x86 to VLIW Translation

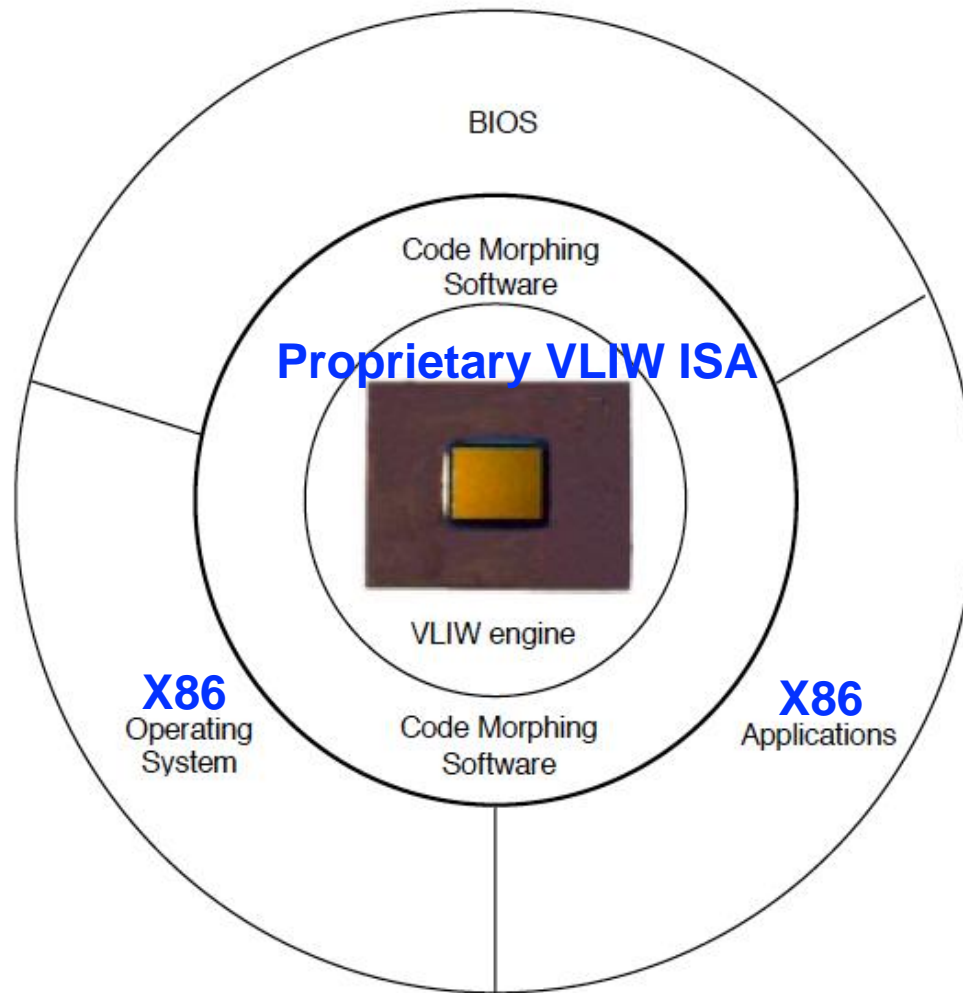
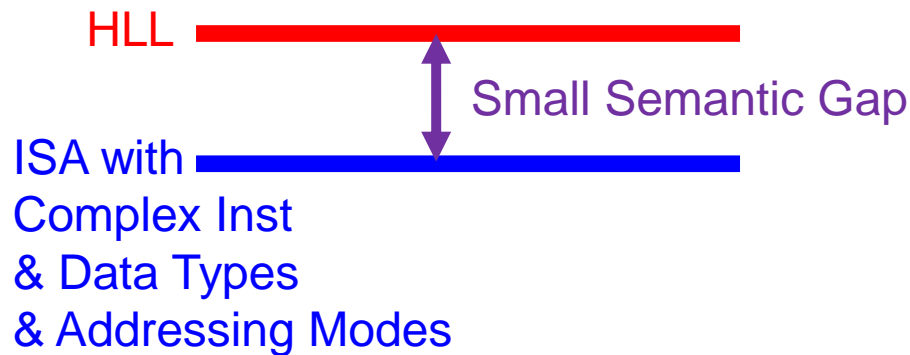


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, “[The Technology Behind Crusoe Processors](#),” Transmeta White Paper 2000.

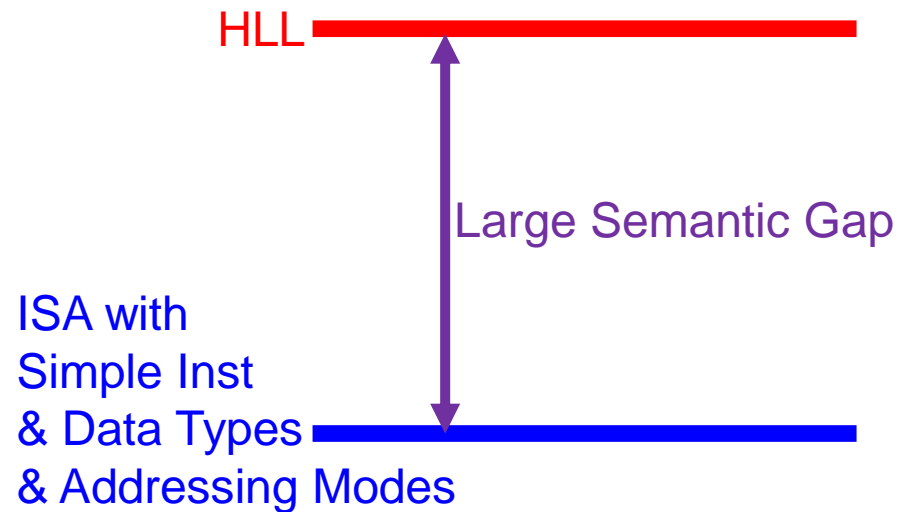
Recall: Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)



HW Control Signals

Easier mapping of HLL to ISA
Less work for software designer
More work for hardware designer
Optimization burden on HW

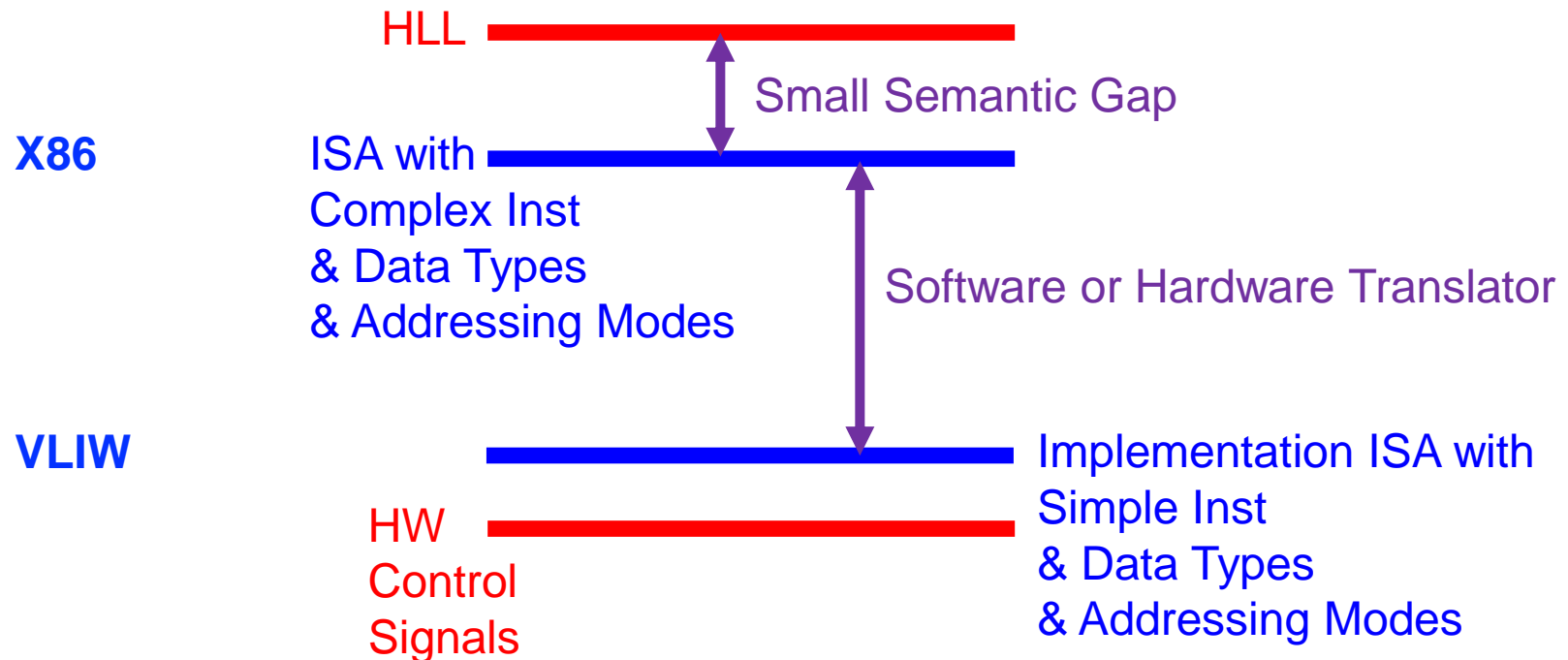


HW Control Signals

Harder mapping of HLL to ISA
More work for software designer
Less work for hardware designer
Optimization burden on SW

Recall: How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different “implementation” ISA



Transmeta: x86 to VLIW Translation

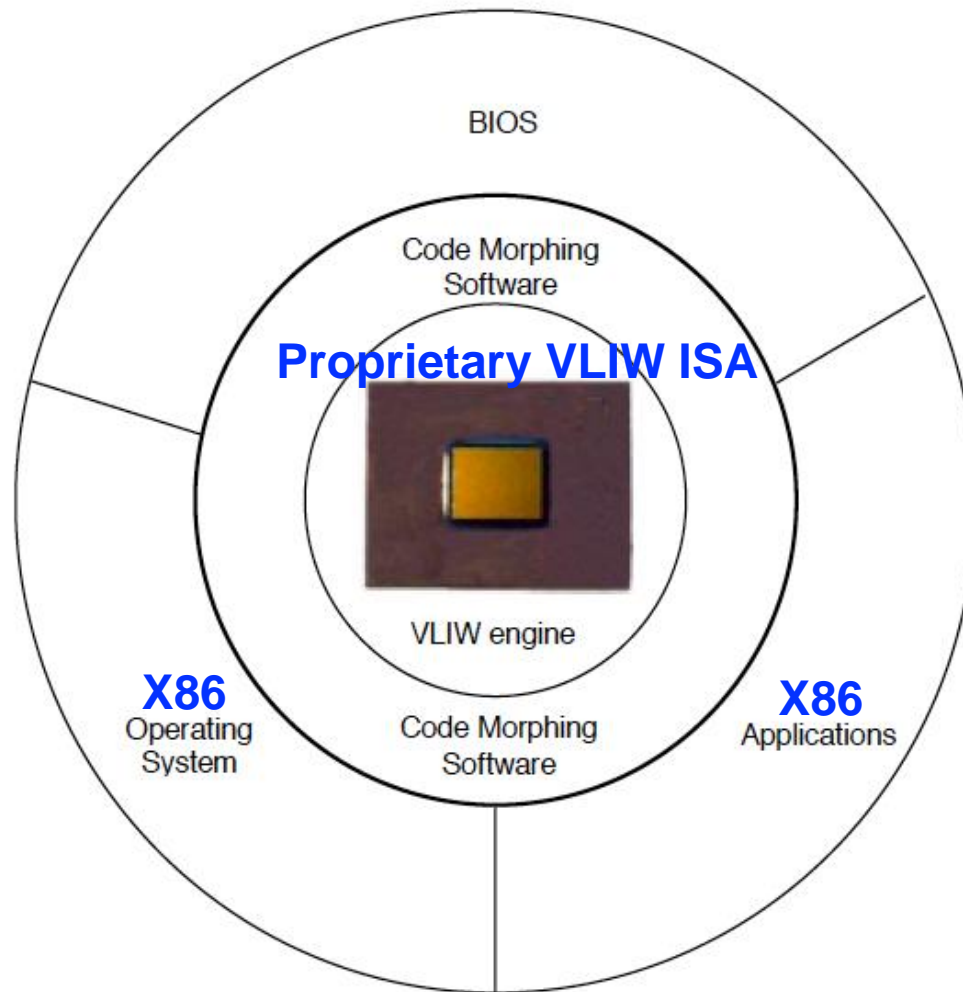


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, “[The Technology Behind Crusoe Processors](#),” Transmeta White Paper 2000.

Another Example: Rosetta 2 Binary Translator

Rosetta 2 [\[edit \]](#)

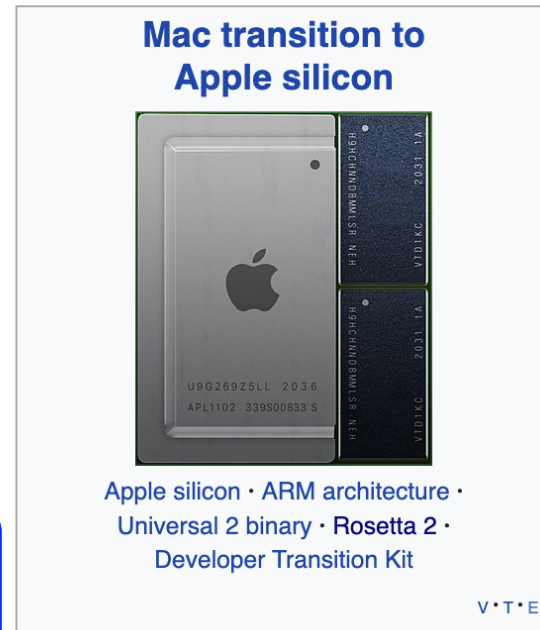
In 2020, Apple announced Rosetta 2 would be bundled with [macOS Big Sur](#), to aid in the [Mac transition to Apple silicon](#). The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.^{[2][8]}

In addition to the [just-in-time](#) (JIT) translation support, Rosetta 2 offers [ahead-of-time compilation](#) (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.^[9]

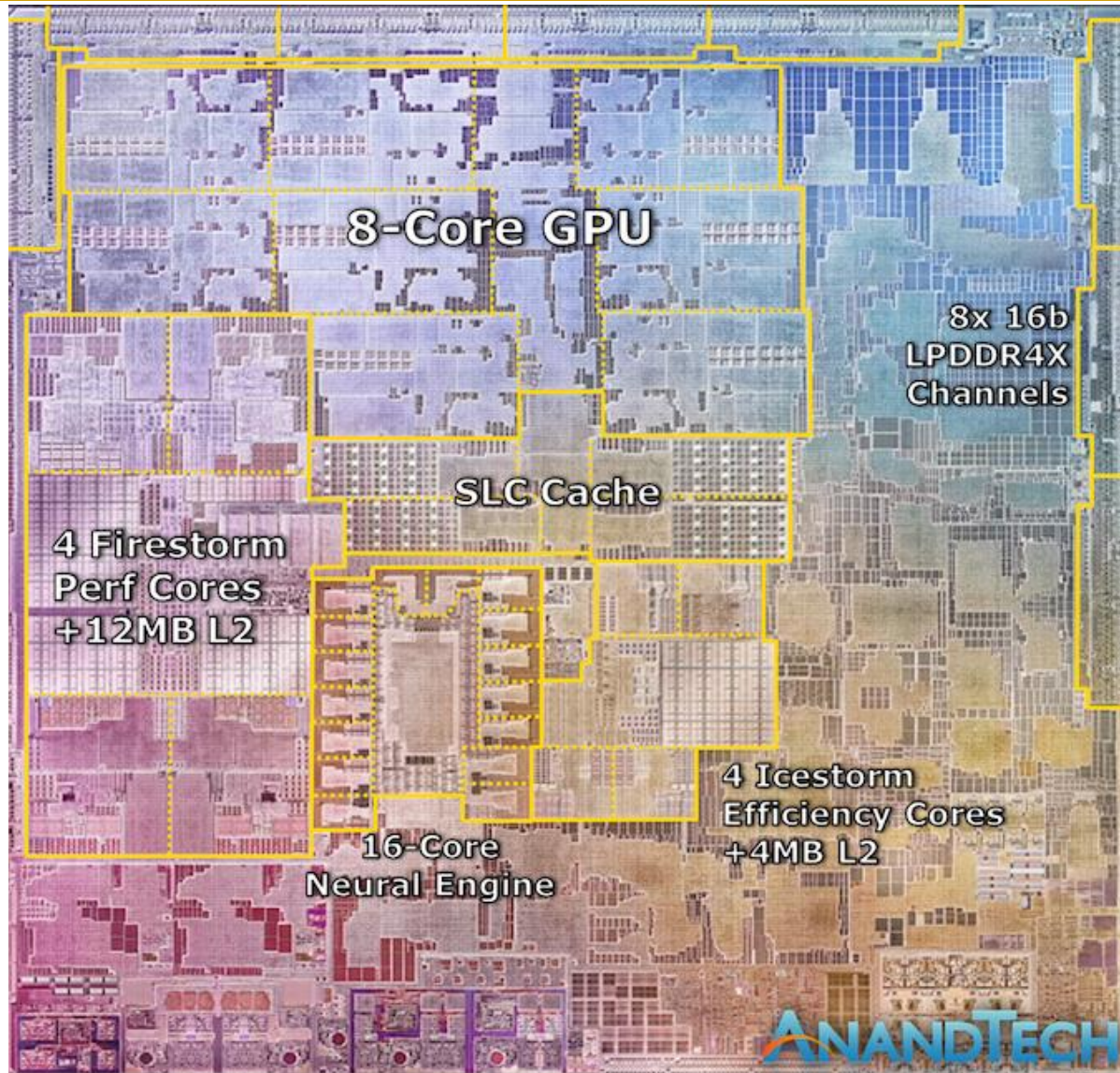
Rosetta 2's performance has been praised greatly.^{[10][11]} In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 [memory ordering](#) in Apple M1 SOC.^[12]

Although Rosetta 2 works for most software, some software doesn't work at all^[13] or is reported to be "sluggish".^[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes [assembly language](#) code, or that generates [machine code](#)), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.^[15]



Another Example: Rosetta 2 Binary Translator



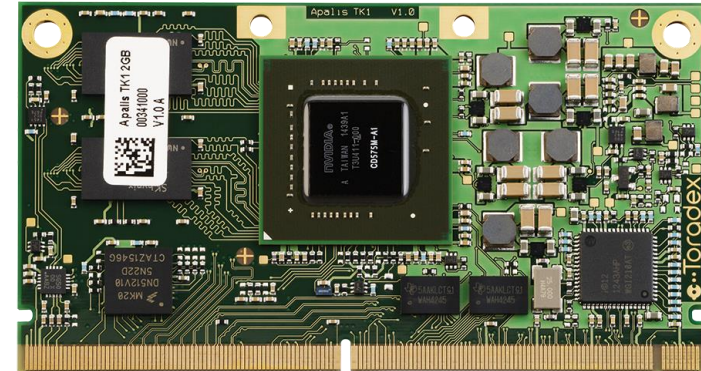
Apple M1,
2021

Another Example: NVIDIA Denver

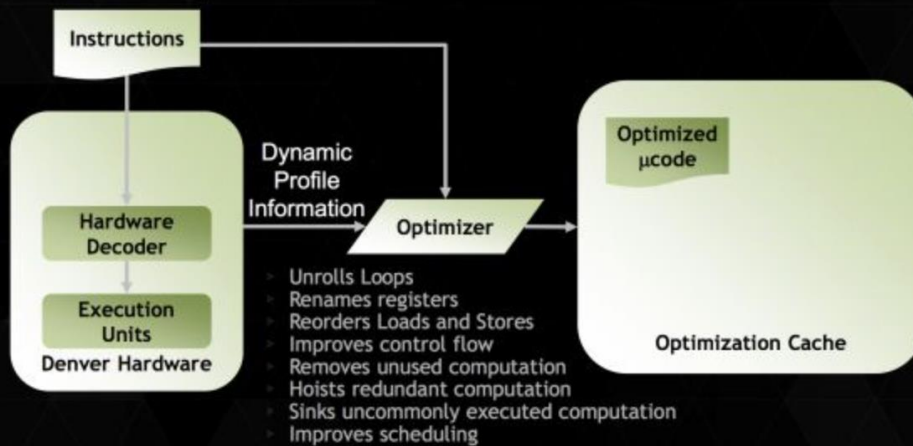
The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.



DYNAMIC CODE OPTIMIZATION OPTIMIZE ONCE, USE MANY TIMES



The DCO system employed in the Denver CPU is codesigned software that extends ideas from prior system-level binary translators.² The primary function is to execute the user's code. The secondary function is to profile execution, create, optimize, and manage regions of tens to thousands of ARM instructions to form equivalent microcode-optimized regions that execute efficiently on the underlying microarchitecture.

More on NVIDIA Denver Code Optimizer

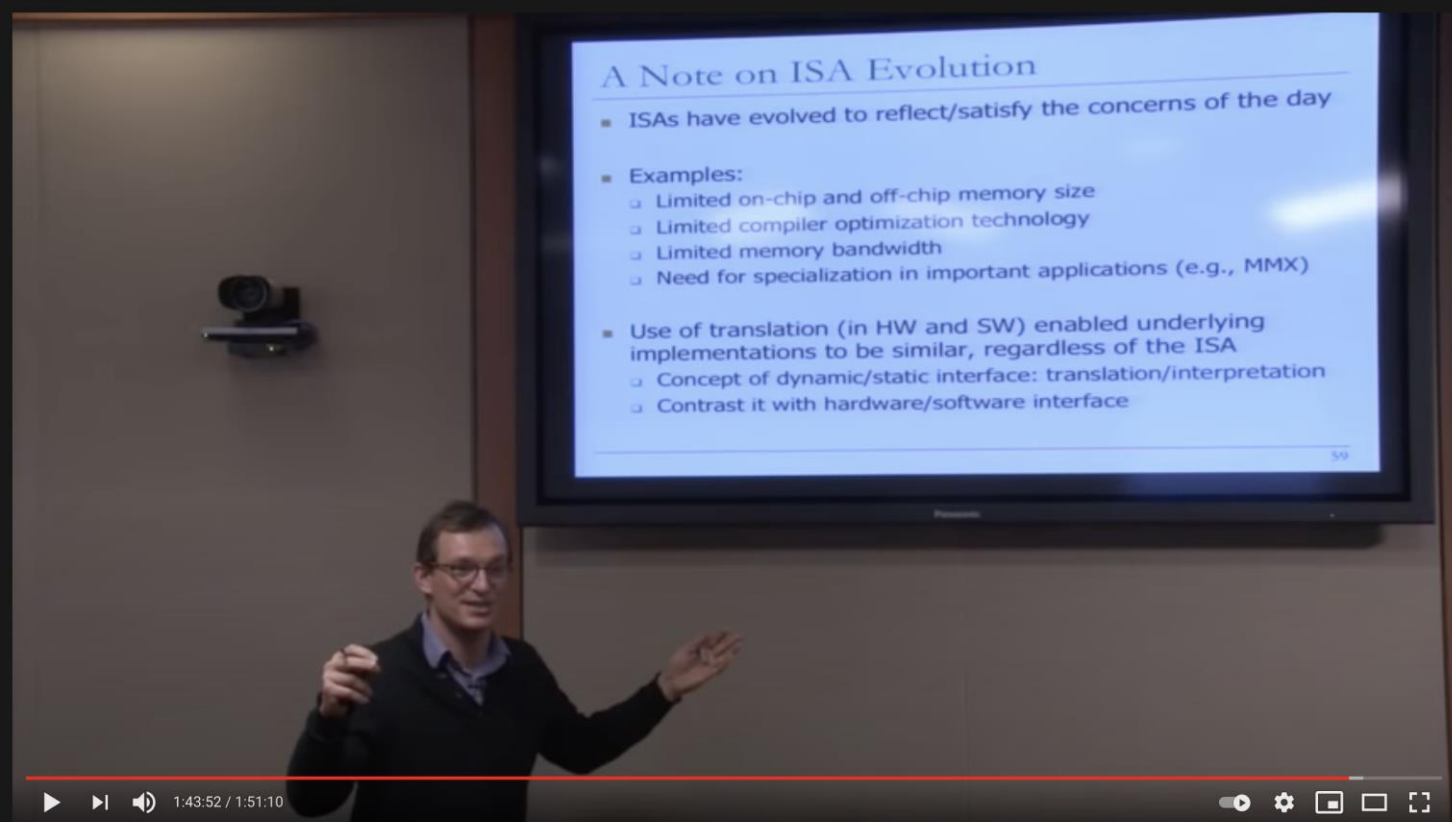
DENVER: NVIDIA'S FIRST 64-BIT ARM PROCESSOR

NVIDIA'S FIRST 64-BIT ARM PROCESSOR, CODE-NAMED DENVER, LEVERAGES A HOST OF NEW TECHNOLOGIES, SUCH AS DYNAMIC CODE OPTIMIZATION, TO ENABLE HIGH-PERFORMANCE MOBILE COMPUTING. IMPLEMENTED IN A 28-NM PROCESS, THE DENVER CPU CAN ATTAIN CLOCK SPEEDS OF UP TO 2.5 GHZ. THIS ARTICLE OUTLINES THE DENVER ARCHITECTURE, DESCRIBES ITS TECHNOLOGICAL INNOVATIONS, AND PROVIDES RELEVANT COMPARISONS AGAINST COMPETING MOBILE PROCESSORS.

Codesigning a hardware processor with a DCO software system creates both additional validation exposure and benefits. The DCO system can be upgraded in the field to address functionality, performance, or security issues.

The Denver hardware decoder provides a mechanism for periodically profiling recently taken branches. This branch history is moved into a shared buffer that can be processed from other cores, thereby minimizing the latency of the interruption. The DCO system will then run a thread that uses this profile to evaluate the dynamic properties of code executing and to assemble a picture of which code regions are hottest across all the processors. On finding sufficiently hot code, the DCO system will begin an optimization process to turn this input ARM code into a microcode execution region. The optimization process uses well-known traditional³ and more speculative compiler techniques to reduce work and increase efficiency of execution on the underlying skewed pipeline. To keep the latency of interruptions to a minimum, the optimizer thread is time-sliced with ARM execution (if any) and runs in a mode that can be quickly interrupted.

There Is A Lot More to Cover on ISAs



A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples:
 - Limited on-chip and off-chip memory size
 - Limited compiler optimization technology
 - Limited memory bandwidth
 - Need for specialization in important applications (e.g., MMX)
- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
 - Concept of dynamic/static interface: translation/interpretation
 - Contrast it with hardware/software interface

59

Lecture 3. ISA Tradeoffs - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

44,973 views • Jan 24, 2015

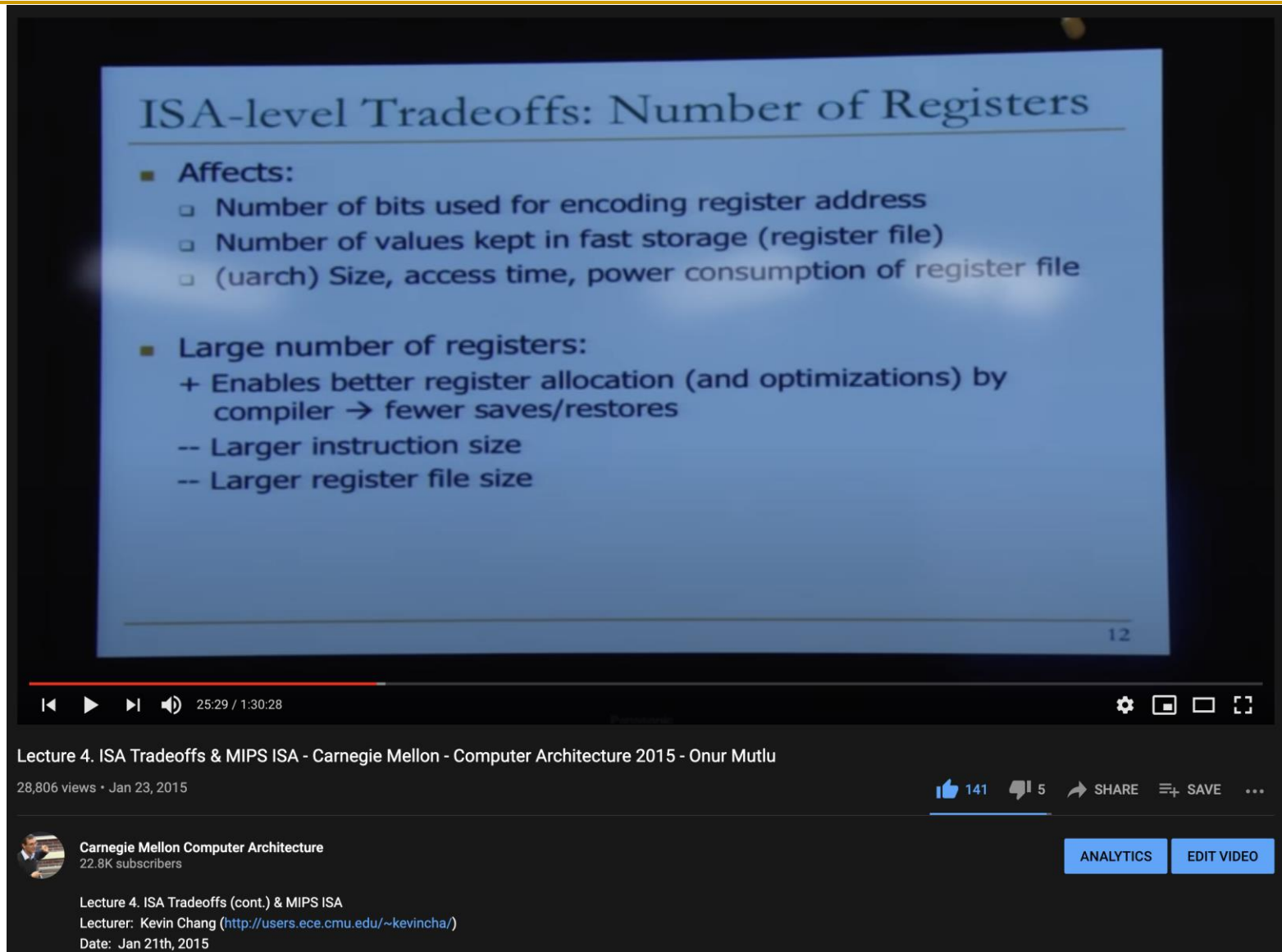
Carnegie Mellon Computer Architecture
22.8K subscribers

Lecture 3. ISA Tradeoffs
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: Jan 16th, 2015

276 5 SHARE SAVE ...

ANALYTICS EDIT VIDEO

There Is A Lot More to Cover on ISAs



The video player displays a slide with the title "ISA-level Tradeoffs: Number of Registers". The slide content is as follows:

- **Affects:**
 - Number of bits used for encoding register address
 - Number of values kept in fast storage (register file)
 - (uarch) Size, access time, power consumption of register file
- **Large number of registers:**
 - + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
 - Larger instruction size
 - Larger register file size

The video player interface includes a progress bar at 25:29 / 1:30:28, a title "Lecture 4. ISA Tradeoffs & MIPS ISA - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu", 28,806 views, and a date of Jan 23, 2015. The channel is "Carnegie Mellon Computer Architecture" with 22.8K subscribers. The video title is "Lecture 4. ISA Tradeoffs (cont.) & MIPS ISA", the lecturer is Kevin Chang, and the date is Jan 21th, 2015.

Detailed Lectures on ISAs & ISA Tradeoffs

■ Computer Architecture, Spring 2015, Lecture 3

- ISA Tradeoffs (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=QKdiZSfwg-g&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=3>

■ Computer Architecture, Spring 2015, Lecture 4

- ISA Tradeoffs & MIPS ISA (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4>

■ Computer Architecture, Spring 2015, Lecture 2

- Fundamental Concepts and ISA (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=2>

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)

Readings for Today

■ Required

- H. T. Kung, “[Why Systolic Architectures?](#),” IEEE Computer 1982.

■ Recommended

- Jouppi et al., “[In-Datacenter Performance Analysis of a Tensor Processing Unit](#)”, ISCA 2017.

Readings for Next Week

■ Required

- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

■ Recommended

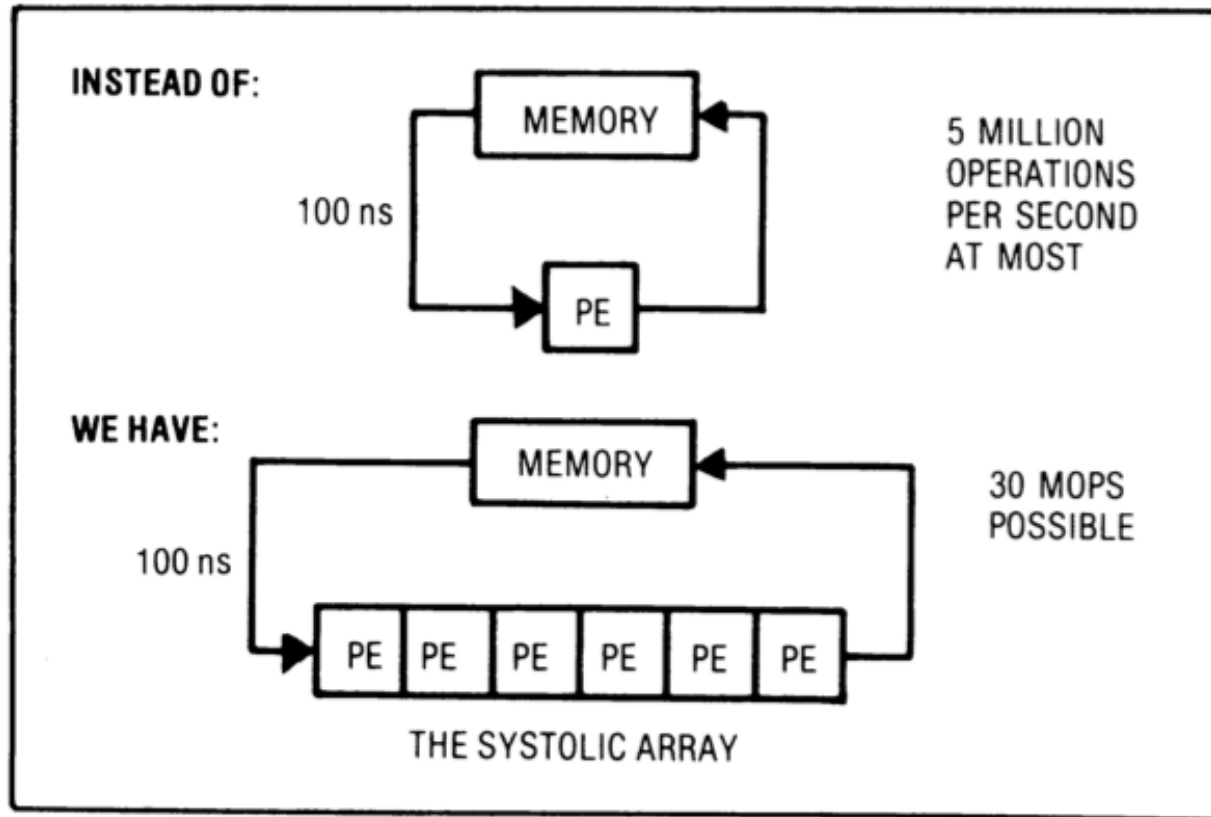
- Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro 1996.

Systolic Arrays

Systolic Arrays: Motivation

- Goal: design an accelerator that has
 - Simple, regular design (keep # unique parts small and regular)
 - High concurrency → high performance
 - Balanced computation and I/O (memory) bandwidth
- Idea: Replace a single processing element (PE) with a regular array of PEs and carefully orchestrate flow of data between the PEs
 - such that they collectively transform a piece of input data before outputting it to memory
- Benefit: Maximizes computation done on a single piece of data element brought from memory

Systolic Arrays



Memory: heart
Data: blood
PEs: cells

Memory pulses
data through
PEs

Figure 1. Basic principle of a systolic system.

- H. T. Kung, "[Why Systolic Architectures?](#)," IEEE Computer 1982.

Why Systolic Architectures?

- Idea: Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory
- Similar to blood flow: heart → many cells → heart
 - Different cells “process” the blood
 - Many veins operate simultaneously
 - Can be many-dimensional
- Why? Special purpose accelerators/architectures need
 - Simple, regular design (keep # unique parts small and regular)
 - High concurrency → high performance
 - Balanced computation and I/O (memory) bandwidth

Systolic Architectures

- Basic principle: Replace a single PE with a **regular array of PEs** and **carefully orchestrate flow of data** between the PEs
 - Balance computation and memory bandwidth

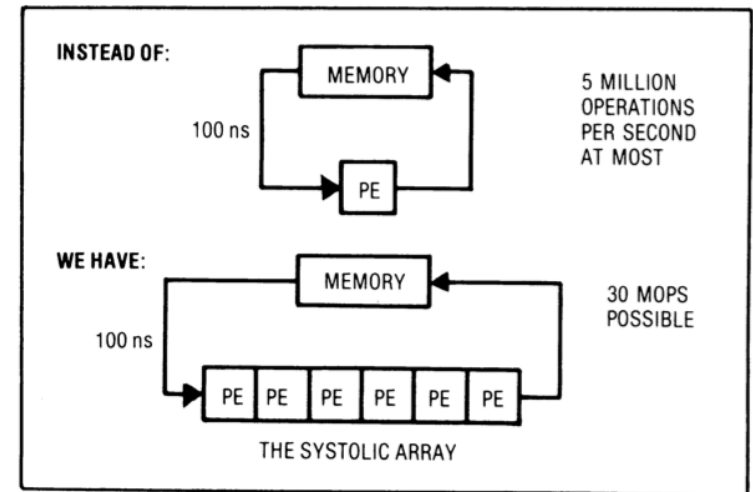


Figure 1. Basic principle of a systolic system.

- Differences from pipelining:
 - These are individual PEs
 - Array structure can be non-linear and multi-dimensional
 - PE connections can be multidirectional (and different speed)
 - PEs can have local memory and execute kernels (rather than a piece of the instruction)

Systolic Computation Example

■ Convolution

- Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
- Many **image processing** tasks
- **Machine learning**: up to hundreds of **convolutional layers** in Convolutional Neural Networks (CNN)

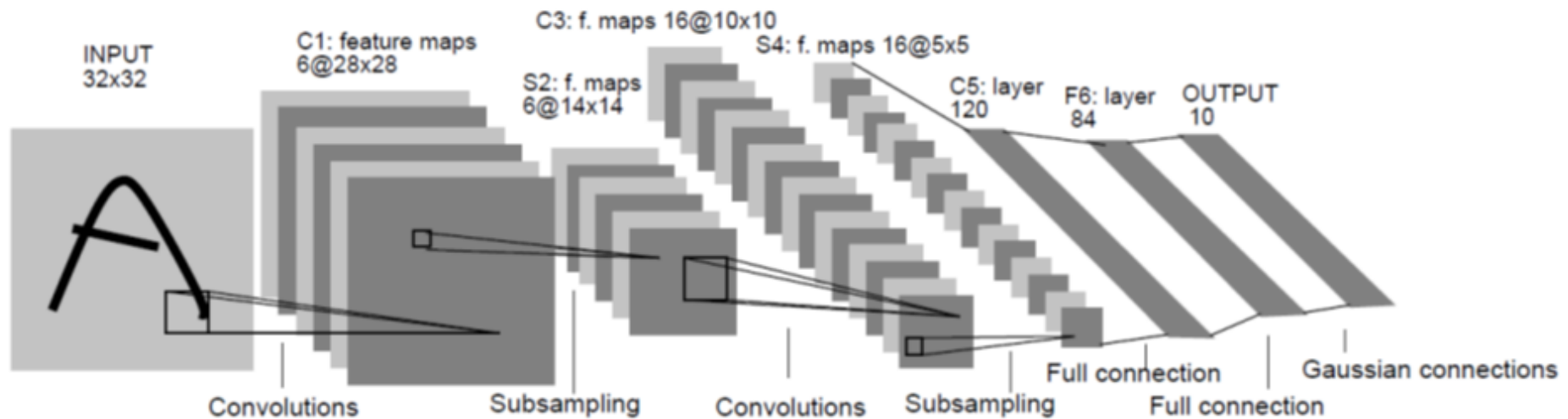
Given the sequence of weights $\{w_1, w_2, \dots, w_k\}$
and the input sequence $\{x_1, x_2, \dots, x_n\}$,

compute the result sequence $\{y_1, y_2, \dots, y_{n+1-k}\}$
defined by

$$y_i = w_1x_i + w_2x_{i+1} + \dots + w_kx_{i+k-1}$$

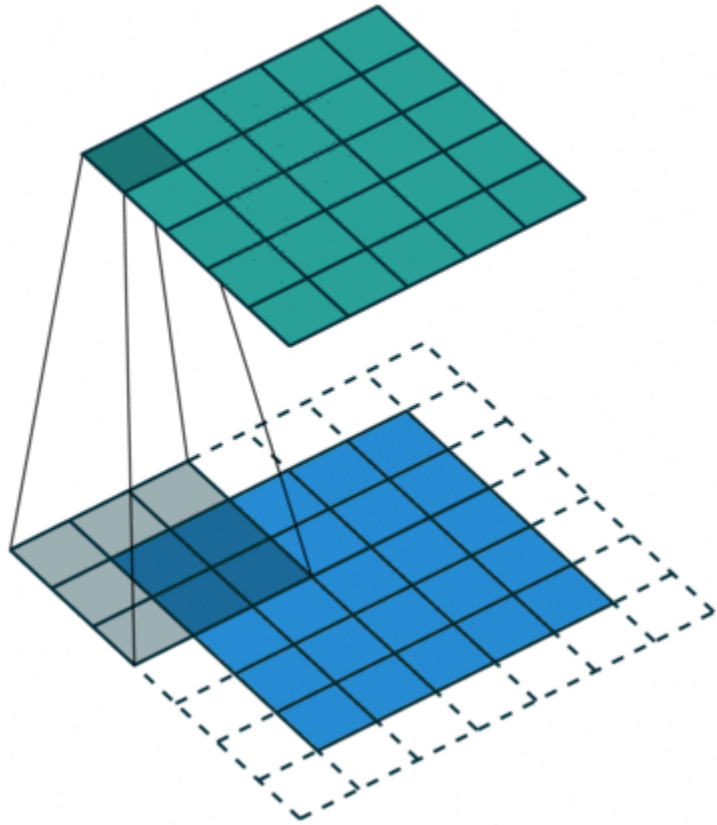
LeNet-5, a Convolutional Neural Network for Hand-Written Digit Recognition

This is a 1024*8 bit input, which will have a truth table of 2^{8196} entries



An Example of 2D Convolution

Output feature map



Input feature map

Structure information

Input: 5*5 (blue)

Kernel (filter): 3*3 (grey)

Output: 5*5 (green)

Computation information

Stride: 1

Padding: 1 (white)

Output Dim = (Input + 2*Padding
- Kernel) / Stride + 1

An Example of 2D Convolution

**Input
Layer**

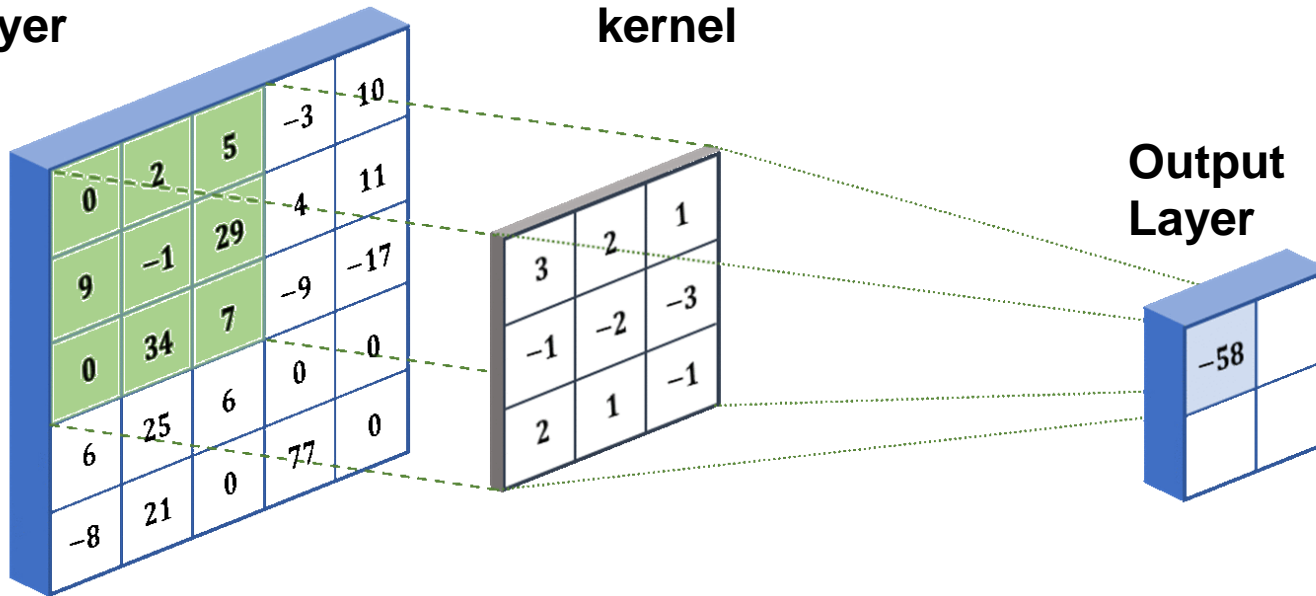
0	2	5	-3	10
9	-1	29	4	11
0	34	7	-9	-17
6	25	6	0	0
-8	21	0	77	0

**CNN
kernel**

3	2	1
-1	-2	-3
2	1	-1

**Output
Layer**

-58	



Convolutional Neural Networks: Demo

[Back to Yann's Home Publications](#)

LeNet-5 Demos

Unusual Patterns

[unusual styles](#)
[weirdos](#)

Invariance

[translation](#) (anim)
[scale](#) (anim)
[rotation](#) (anim)
[squeezing](#) (anim)
[stroke width](#) (anim)

Noise Resistance

[noisy 3 and 6](#)
[noisy 2](#) (anim)
[noisy 4](#) (anim)

Multiple Character

[various stills](#)
[dancing 00](#) (anim)
[dancing 384](#) (anim)

Complex cases

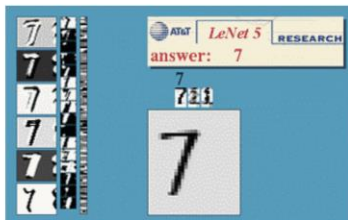
(anim)
[35 -> 53](#)
[12 -> 4 -> 21](#)
[23 -> 32](#)
[30 + noise](#)
[31-51-57-61](#)

LeNet-5, convolutional neural networks

Convolutional Neural Networks are a special kind of multi-layer neural networks. Like almost every other neural networks they are trained with a version of the back-propagation algorithm. Where they differ is in the architecture.

Convolutional Neural Networks are designed to recognize visual patterns directly from pixel images with minimal preprocessing. They can recognize patterns with extreme variability (such as handwritten characters), and with robustness to distortions and simple geometric transformations.

LeNet-5 is our latest convolutional network designed for handwritten and machine-printed character recognition. Here is an example of LeNet-5 in action.



Many more examples are available in the column on the left:

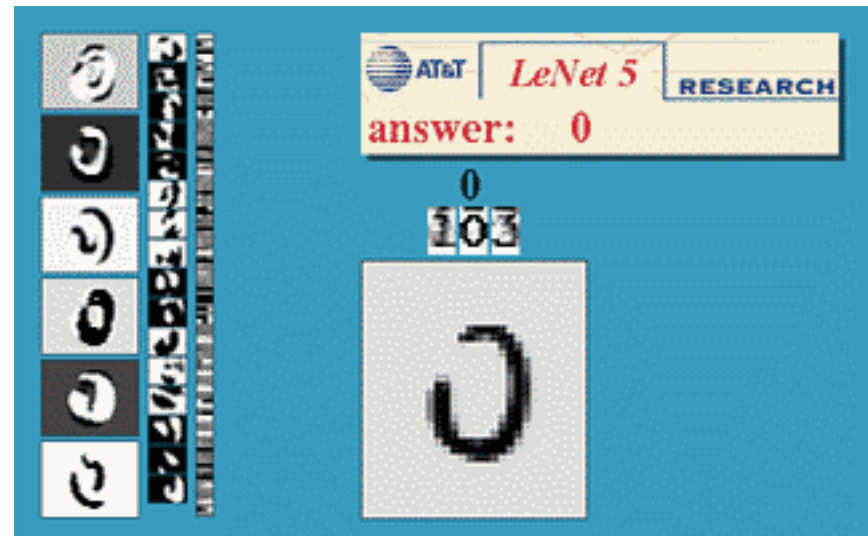
Several papers on LeNet and convolutional networks are available on my [publication page](#):

[LeCun et al., 1998]

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.
Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998.
[ps.gz](#)

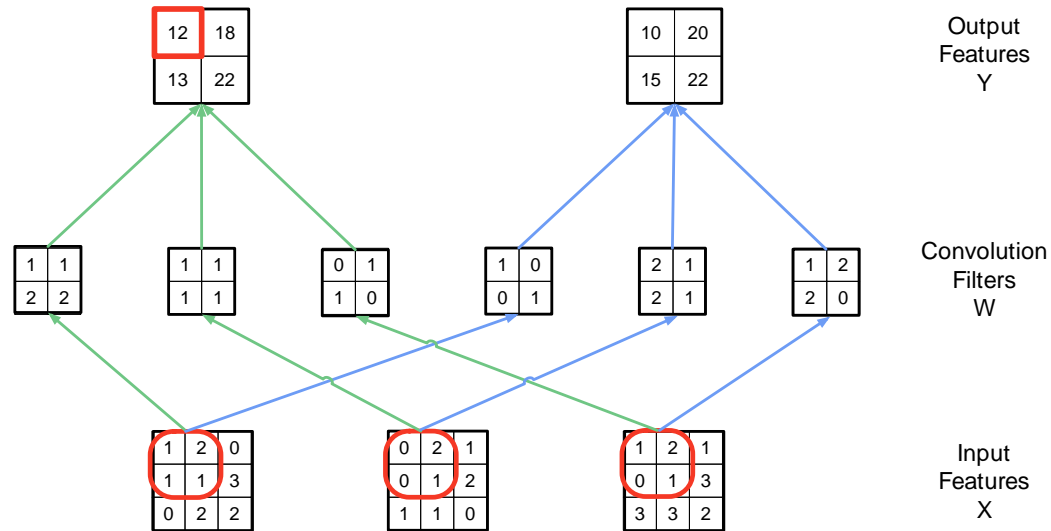
[Bottou et al., 1997]

L. Bottou, Y. LeCun, and Y. Bengio. Global training of



<http://yann.lecun.com/exdb/lenet/index.html>

Implementing a Convolutional Layer with Matrix Multiplication



$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 2 & 2 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 2 & 0 \\ \hline \end{array} * \begin{array}{|c|c|c|c|} \hline 1 & 2 & 1 & 1 \\ \hline 2 & 0 & 1 & 3 \\ \hline 1 & 1 & 0 & 2 \\ \hline 1 & 3 & 2 & 2 \\ \hline 0 & 2 & 0 & 1 \\ \hline 2 & 1 & 1 & 2 \\ \hline 0 & 1 & 1 & 1 \\ \hline 1 & 2 & 1 & 0 \\ \hline 1 & 2 & 0 & 1 \\ \hline 2 & 1 & 1 & 3 \\ \hline 0 & 1 & 3 & 3 \\ \hline 1 & 3 & 3 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 12 & 18 & 13 & 22 \\ \hline 10 & 20 & 15 & 22 \\ \hline \end{array}$$

Convolution Filters W'

Input Features X (unrolled)

Output Features Y

Power of **Convolutions** and **Applied Courses**

- In 2010, Prof. Andreas Moshovos adopted Professor Hwu's ECE498AL Programming Massively Parallel Processors Class
- Several of Prof. Geoffrey Hinton's graduate students took the course
- These students developed the GPU implementation of the Deep CNN that was trained with 1.2M images to win the ImageNet competition

Example: AlexNet (2012)

- AlexNet wins the **ImageNet classification competition** with ~10% points higher accuracy than state-of-the-art
 - Krizhevsky et al., “**ImageNet Classification with Deep Convolutional Neural Networks**”, NIPS 2012.

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called “dropout” that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

Example: GoogLeNet (2014)

- Google improves accuracy by **adding more network layers**
 - From 8 in AlexNet to 22 in GoogLeNet
 - Szegedy et al., “**Going Deeper with Convolutions**”, CVPR 2015.

Going Deeper with Convolutions

Christian Szegedy¹, Wei Liu², Yangqing Jia¹, Pierre Sermanet¹, Scott Reed³,
Dragomir Anguelov¹, Dumitru Erhan¹, Vincent Vanhoucke¹, Andrew Rabinovich⁴

¹Google Inc. ²University of North Carolina, Chapel Hill

³University of Michigan, Ann Arbor ⁴Magic Leap Inc.

¹{szegedy, jia, yq, sermanet, dragomir, dimitru, vanhoucke}@google.com

²wliu@cs.unc.edu, ³reedscott@umich.edu, ⁴arabinovich@magic Leap Inc.

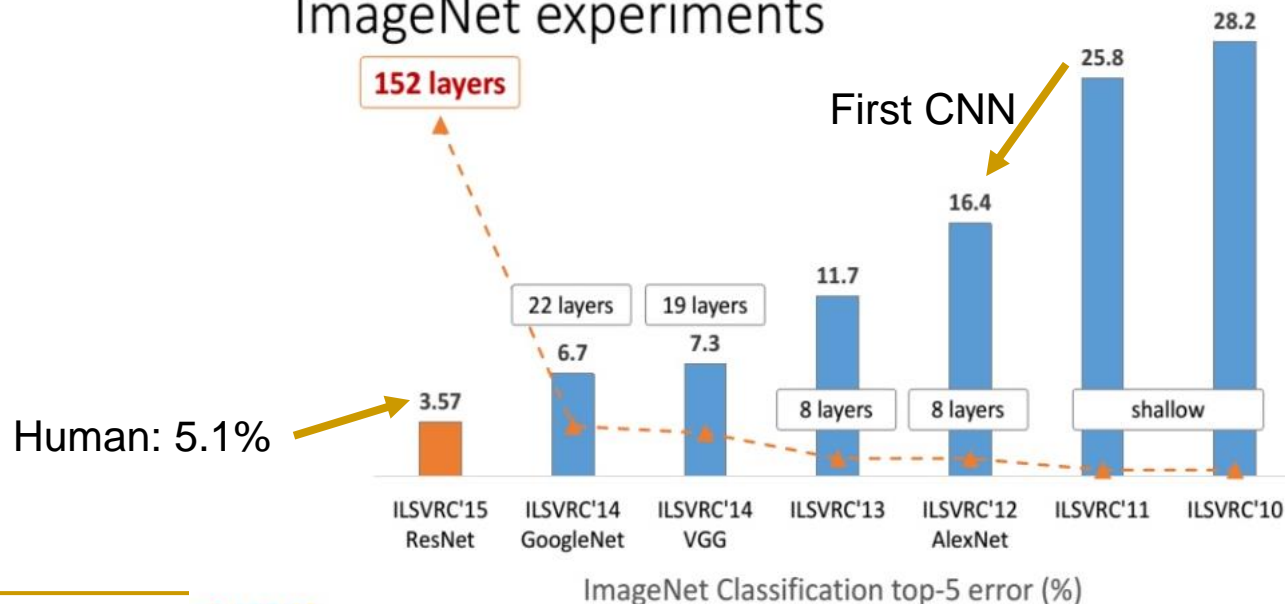
Example: ResNet (2015)

- He et al., “Deep Residual Learning for Image Recognition”, CVPR 2016.

Deep Residual Learning for Image Recognition

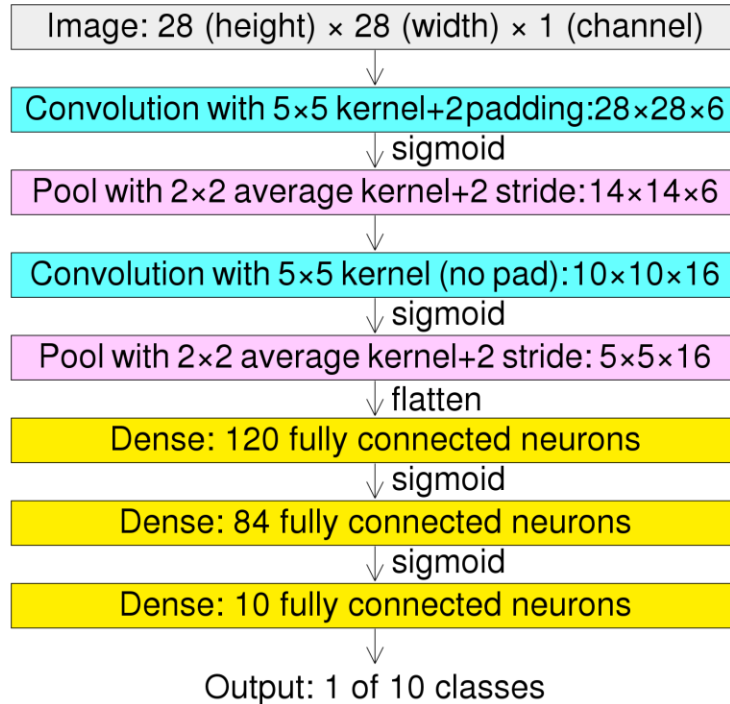
Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

ImageNet experiments

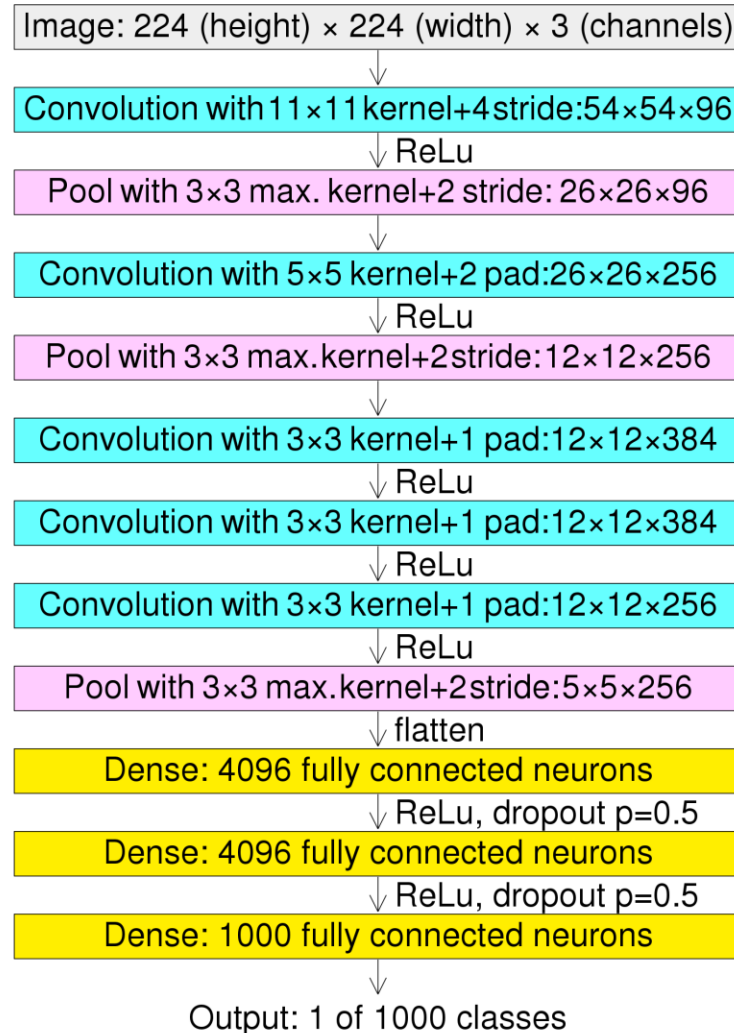


Neural Network Layer Examples

LeNet



AlexNet



Systolic Computation Example: Convolution (I)

■ Convolution

- Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
- Many **image processing** tasks
- **Machine learning**: up to hundreds of **convolutional layers** in Convolutional Neural Networks (CNN)

Given the sequence of weights $\{w_1, w_2, \dots, w_k\}$
and the input sequence $\{x_1, x_2, \dots, x_n\}$,

compute the result sequence $\{y_1, y_2, \dots, y_{n+1-k}\}$
defined by

$$y_i = w_1x_i + w_2x_{i+1} + \dots + w_kx_{i+k-1}$$

Systolic Computation Example: Convolution (II)

- $y_1 = w_1x_1 + w_2x_2 + w_3x_3$
- $y_2 = w_1x_2 + w_2x_3 + w_3x_4$
- $y_3 = w_1x_3 + w_2x_4 + w_3x_5$

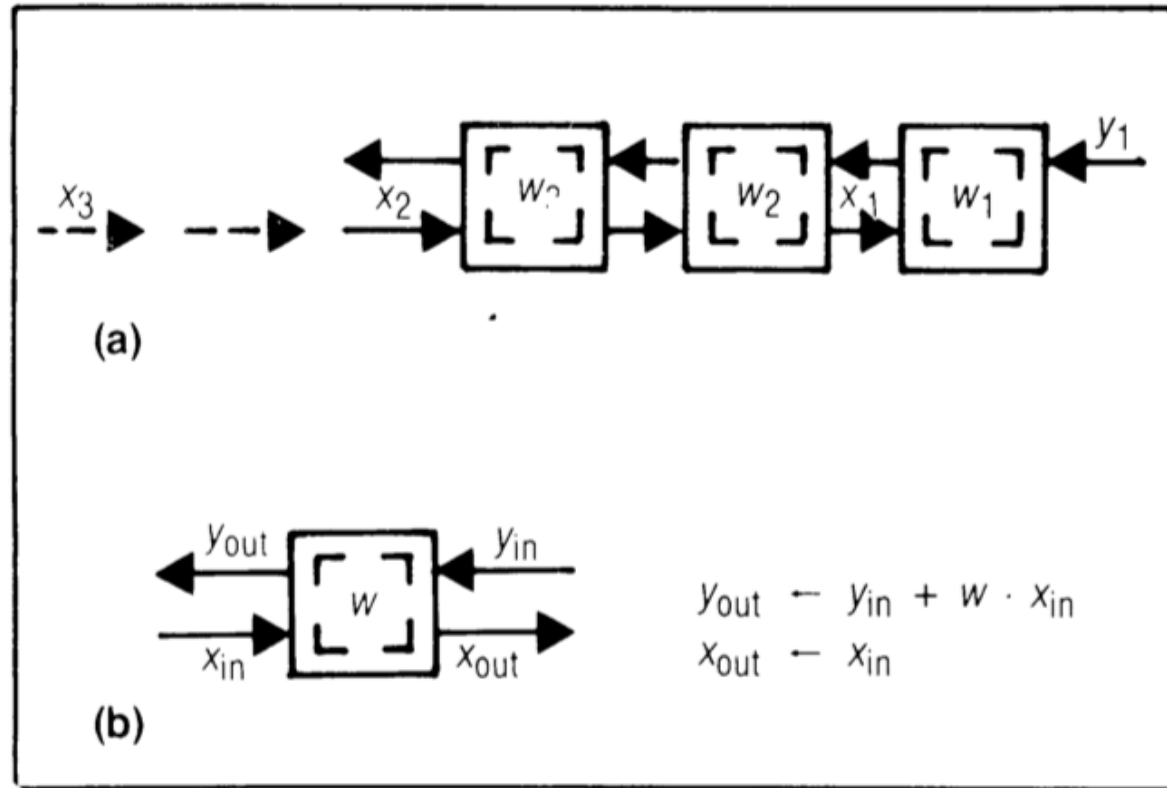


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i 's stay and x_i 's and y_i 's move systolically in opposite directions.

Systolic Computation Example: Convolution (III)

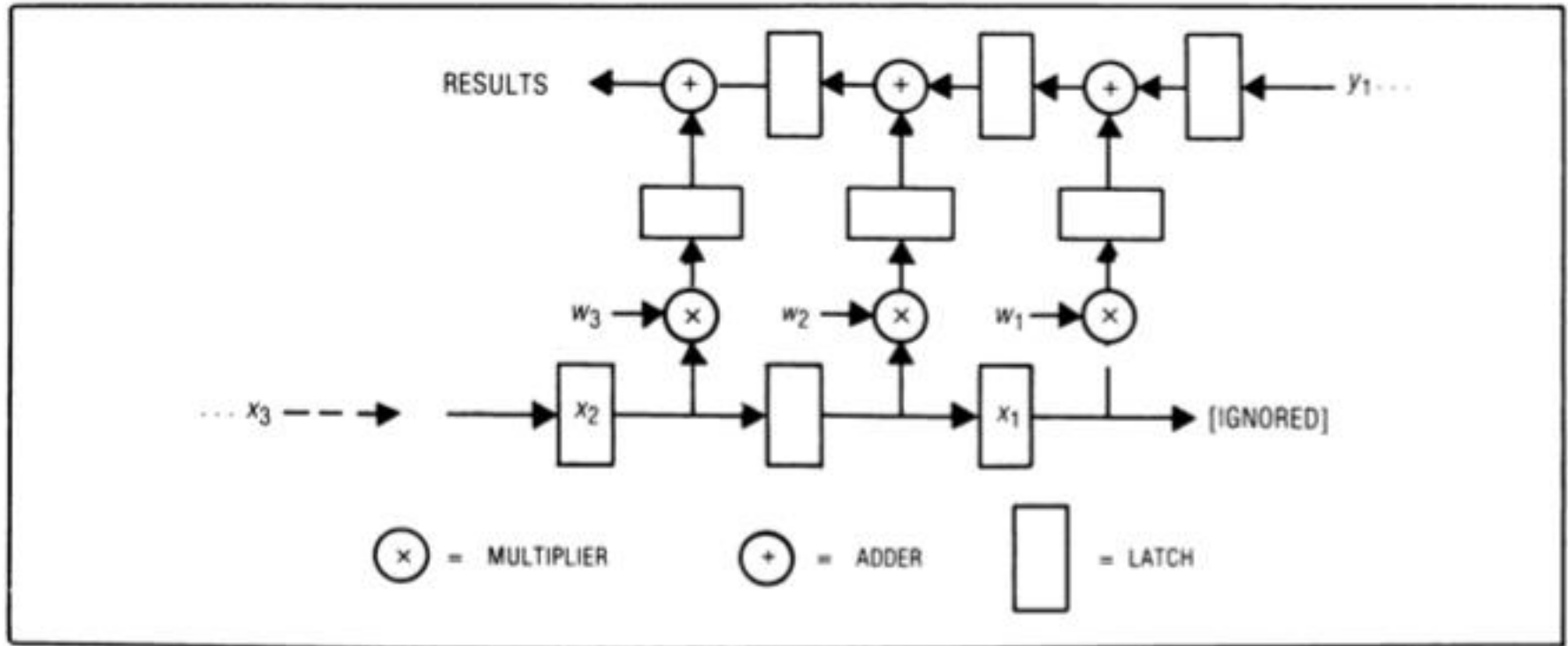


Figure 10. Overlapping the executions of multiply and add in design W1.

- Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions

Systolic Computation Example: Convolution (IV)

- One needs to **carefully orchestrate** when **data elements are input to the array**
- And when **output is buffered**
- This gets more involved when
 - Array dimensionality increases
 - PEs are less predictable in terms of latency

Example 2D Systolic Array Computation

- Multiply two 3x3 matrices (inputs)
 - Keep the final result in PE accumulators

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

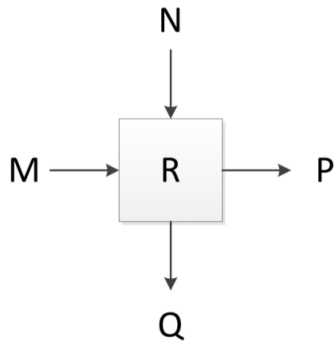
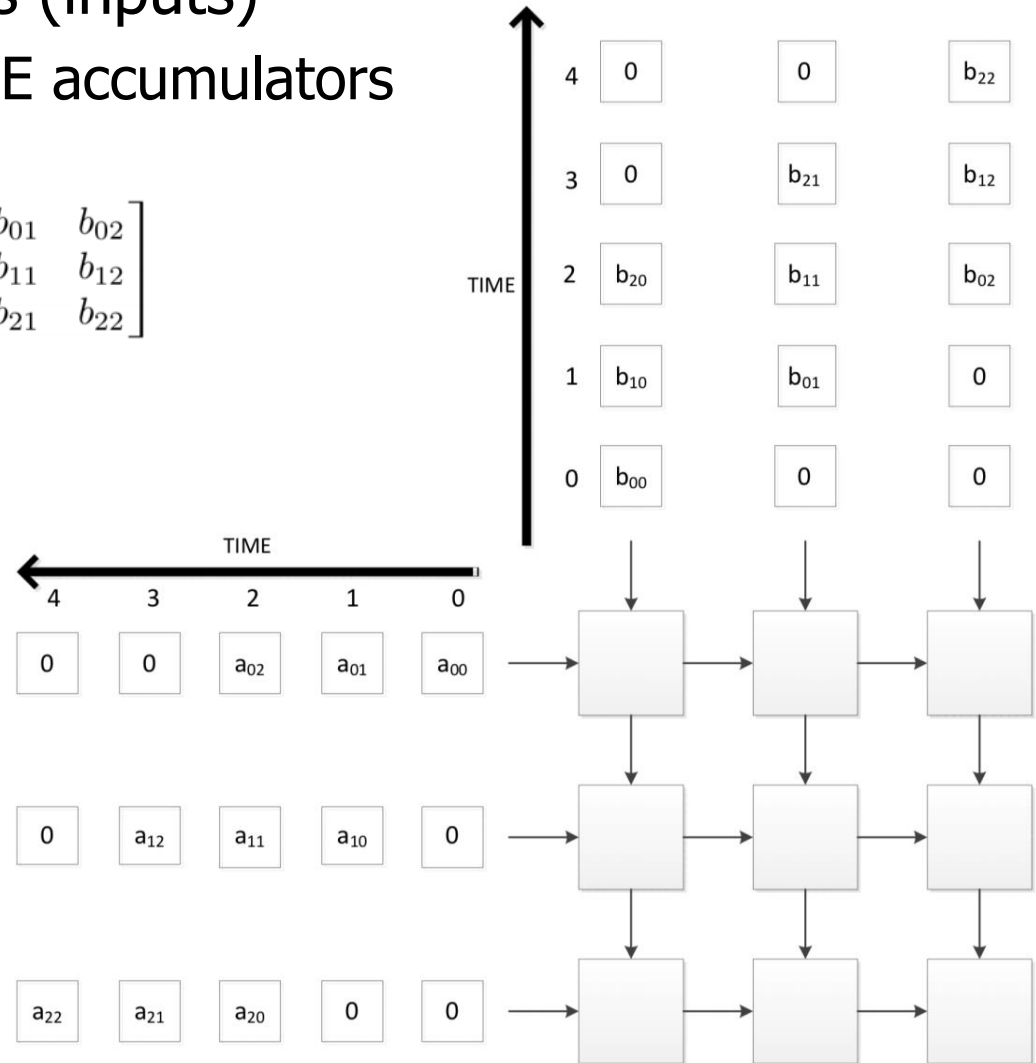


Figure 1: A systolic array processing element

$$\begin{aligned} P &= M \\ Q &= N \\ R &= R + M * N \end{aligned}$$



Two-Dimensional Systolic Arrays

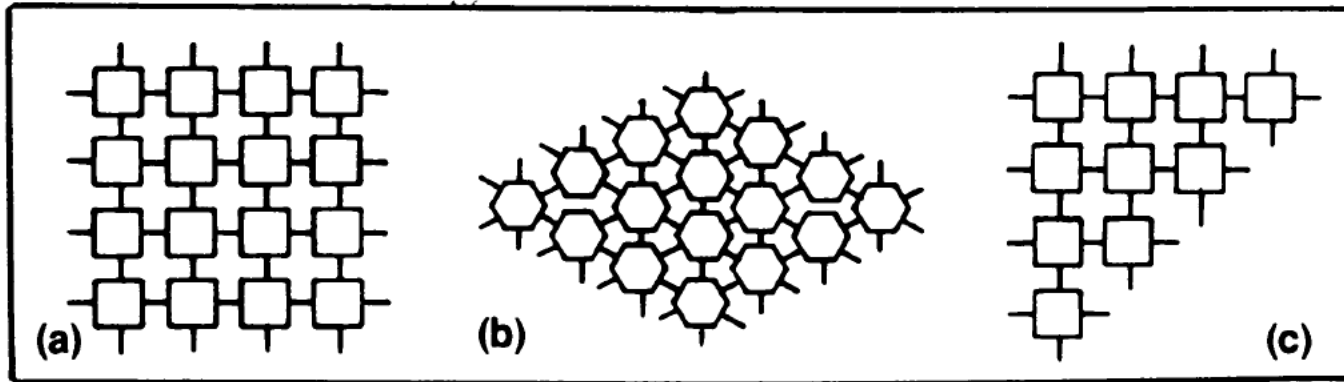


Figure 11. Two-dimensional systolic arrays: (a) type R, (b) type H, and (c) type T.

To a given problem there could be both one- and two-dimensional systolic array solutions. For example, two-dimensional convolution can be performed by a one-dimensional systolic array^{24,25} or a two-dimensional systolic array.⁶ When the memory speed is more than cell speed, two-dimensional systolic arrays such as those depicted in Figure 11 should be used. At each cell cycle, all the I/O ports on the array boundaries can input or output data items to or from the memory; as a result, the available memory bandwidth can be fully utilized. Thus, the choice of a one- or two-dimensional scheme is very dependent on how cells and memories will be implemented.

Combinations

- Systolic arrays can be chained together to form powerful systems
- This systolic array is capable of producing on-the-fly least-squares fit to all the data that has arrived up to any given moment

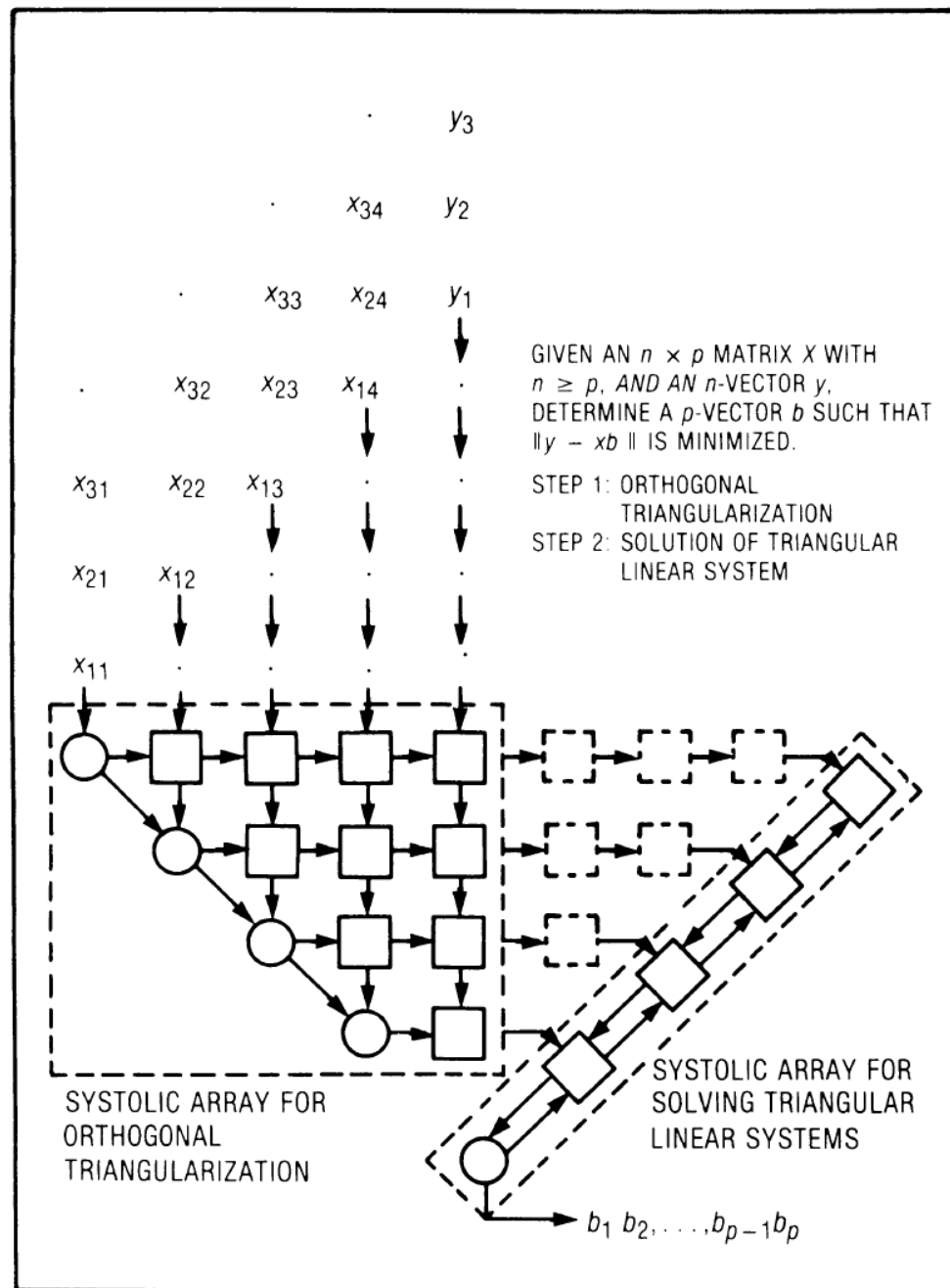


Figure 12. On-the-fly least-squares solutions using one- and two-dimensional systolic arrays, with $p = 4$.

Systolic Arrays: Pros and Cons

■ Advantages:

- ❑ **Principled**: Efficiently makes use of limited memory bandwidth, balances computation to I/O bandwidth availability
- ❑ **Specialized** (computation needs to fit PE organization/functions)
 - improved **efficiency, simple design, high concurrency/performance**
 - good to do **more with less memory bandwidth** requirement

■ Downside:

- ❑ **Specialized**
 - **not generally applicable** because computation needs to fit the PE functions/organization

More Programmability in Systolic Arrays

- Each PE in a systolic array
 - Can store multiple “weights”
 - Weights can be selected on the fly
 - Eases implementation of, e.g., adaptive filtering
- Taken further
 - Each PE can have its own data and instruction memory
 - Data memory → to store partial/temporary results, constants
 - Leads to **stream processing, pipeline parallelism**
 - More generally, **staged execution**

Pipeline-Parallel (Pipelined) Programs

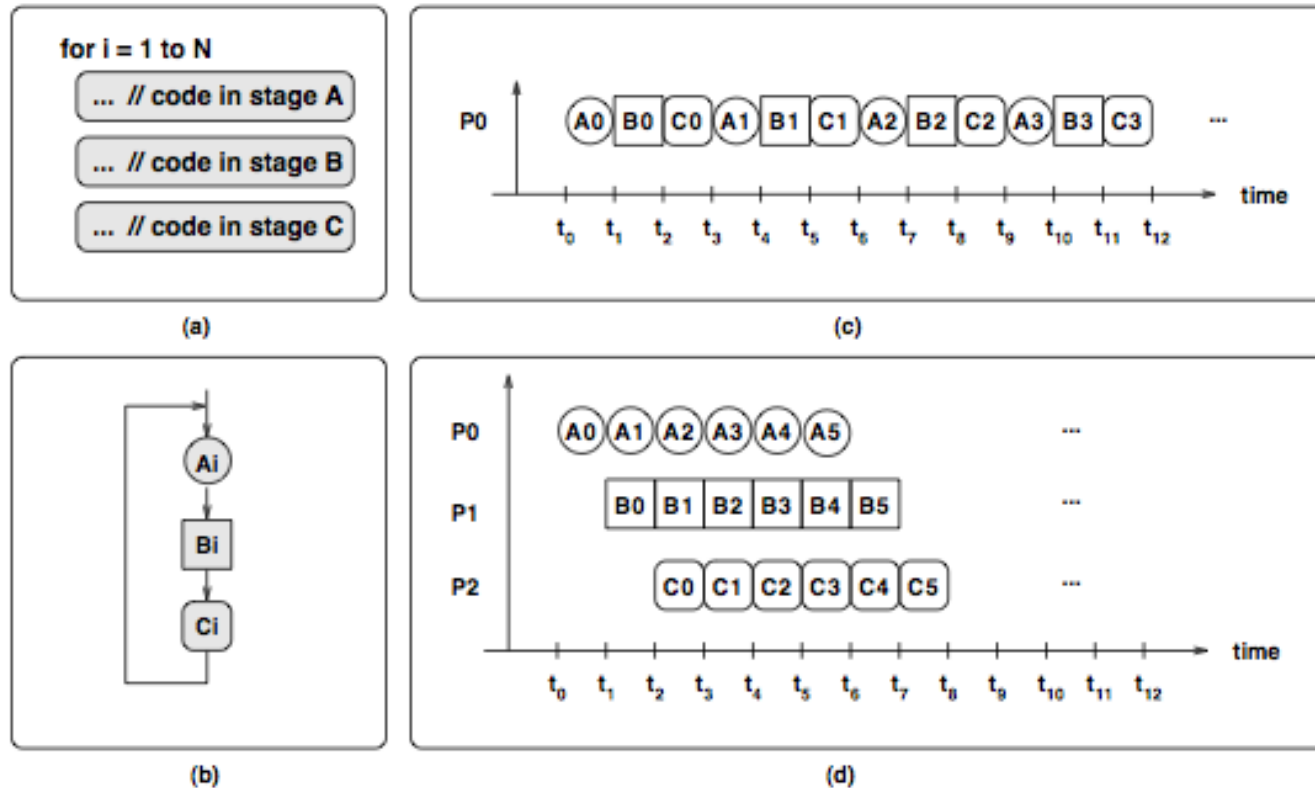
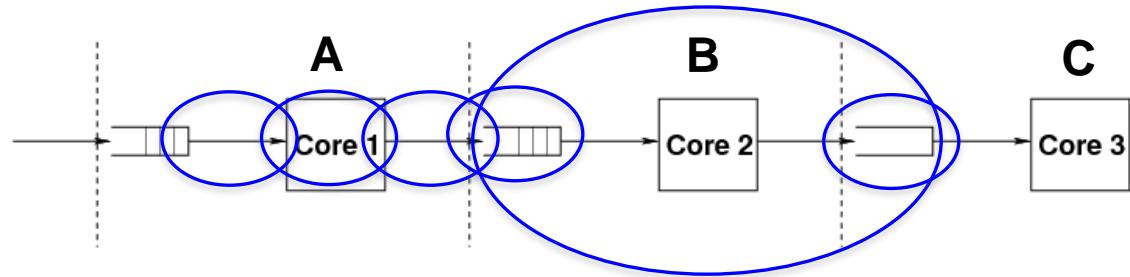


Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises A_i , B_i , C_i . (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

Stages of Pipelined Programs

- Loop iterations are divided into **code segments called stages**
- Threads execute stages on different cores

```
loop {  
  Compute1  A  
  Compute2  B  
  Compute3  C  
}
```



Pipelined File Compression Example

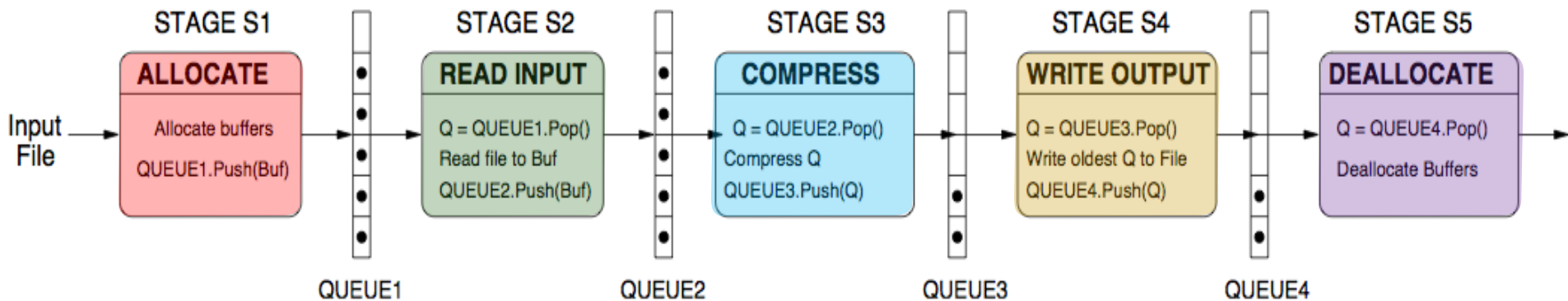


Figure 3. File compression algorithm executed using pipeline parallelism

Systolic Array: Advantages & Disadvantages

■ Advantages

- ❑ Makes **multiple uses of each data item** → reduced need for fetching/refetching → better use of memory bandwidth
- ❑ **High concurrency**
- ❑ Regular design (both data and control flow)

■ Disadvantages

- ❑ **Not good at exploiting irregular parallelism**
- ❑ Relatively special purpose → need software, programmer support to be a general purpose model

Example Systolic Array: The WARP Computer

- HT Kung, CMU, 1984-1988
- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks
- Annaratone et al., “Warp Architecture and Implementation,” ISCA 1986.
- Annaratone et al., “The Warp Computer: Architecture, Implementation, and Performance,” IEEE TC 1987.

The WARP Computer

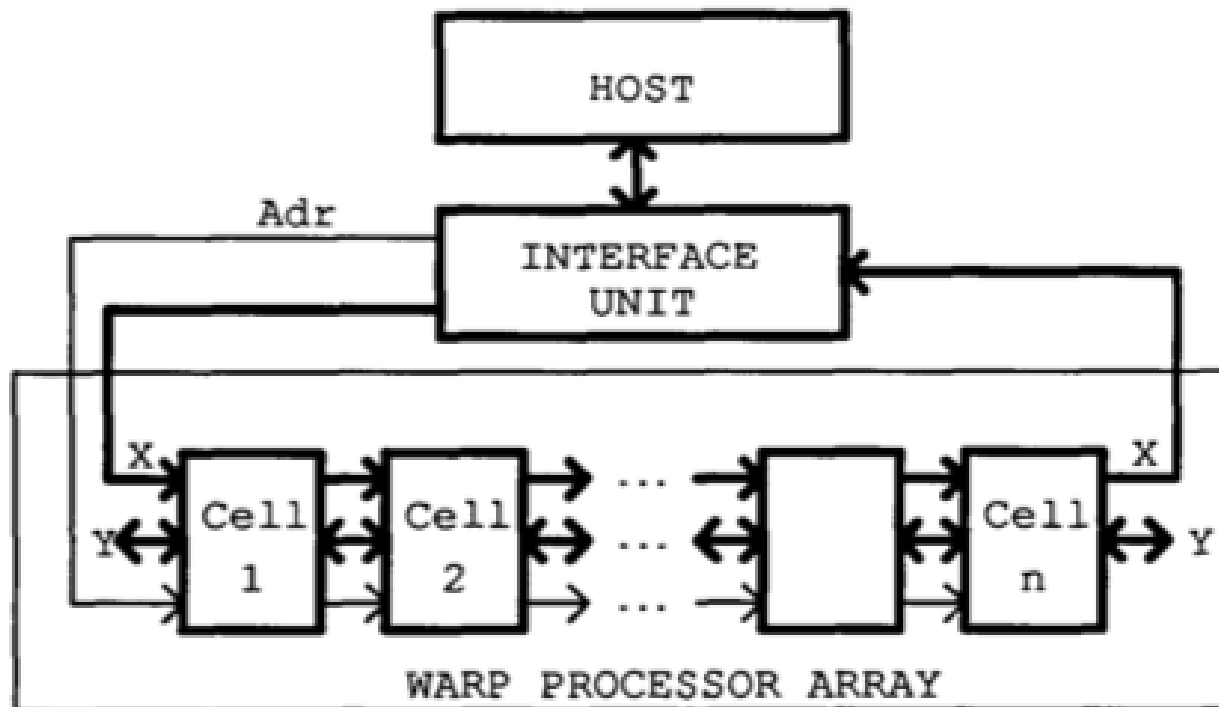


Figure 1: Warp system overview

The WARP Cell

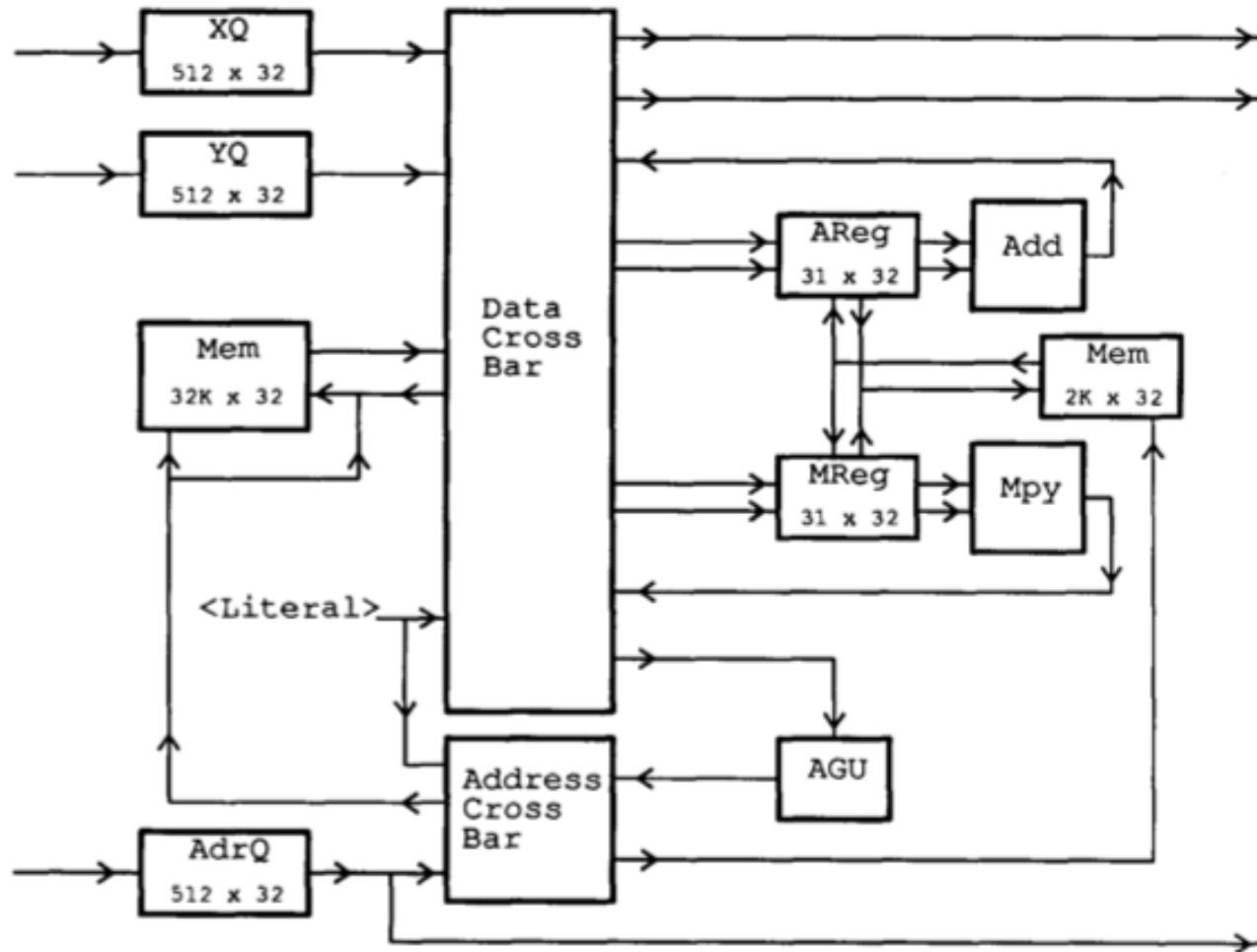


Figure 2: Warp cell data path

An Example Modern Systolic Array: TPU (I)

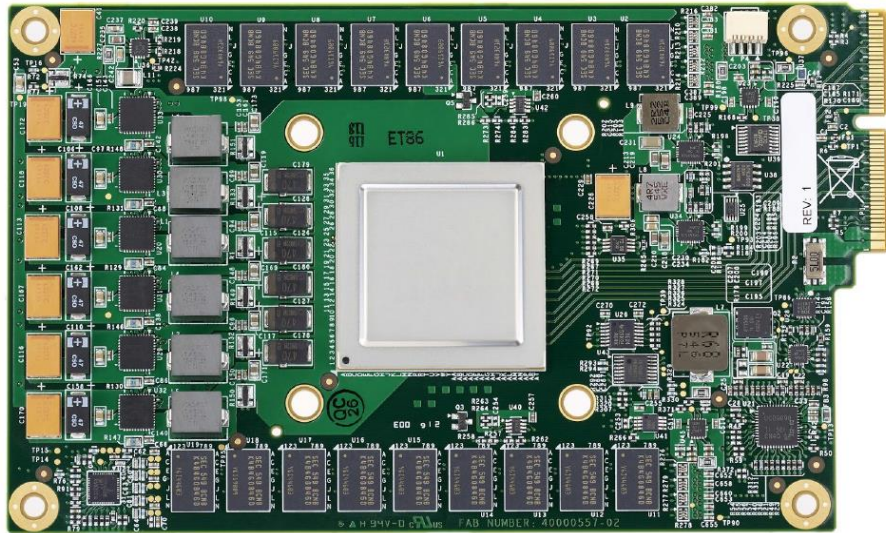


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

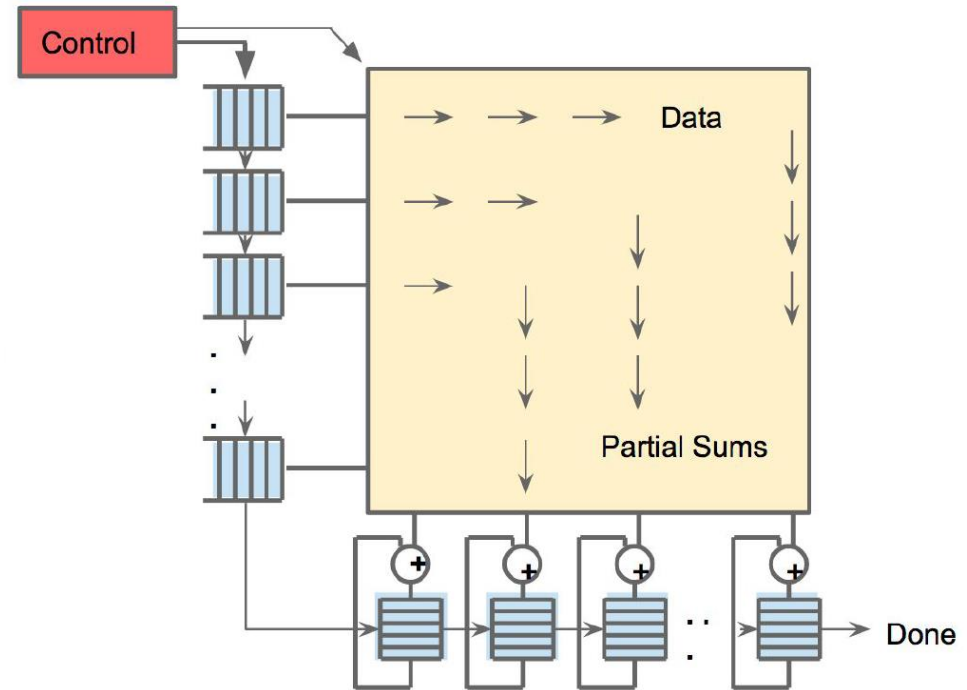
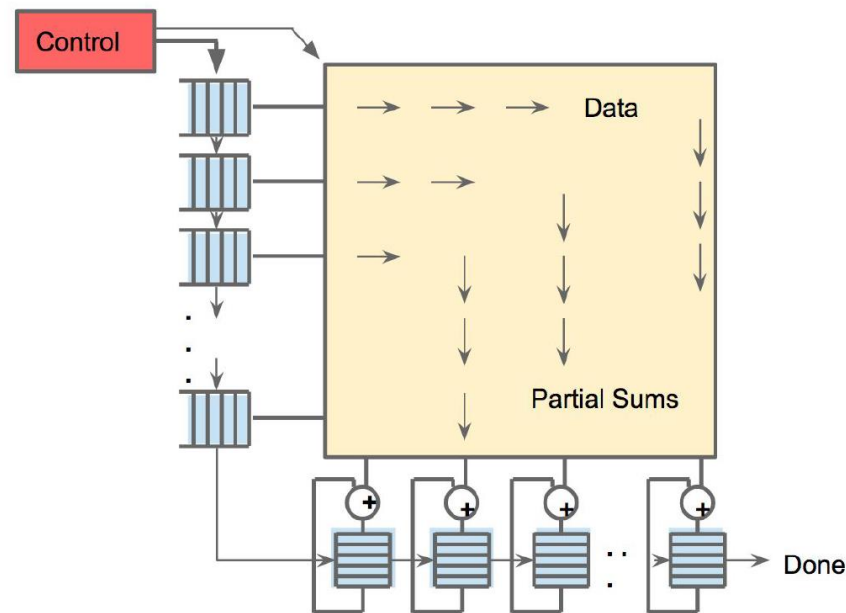


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

Jouppi et al., “In-Datacenter Performance Analysis of a Tensor Processing Unit”, ISCA 2017.

An Example Modern Systolic Array: TPU (II)

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer [Kun80][Ram91][Ovt15b]. Figure 4 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. The weights are preloaded, and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.



Jouppi et al., “In-Datacenter Performance Analysis of a Tensor Processing Unit”, ISCA 2017.

Recall: Example 2D Systolic Array Computation

- Multiply two 3x3 matrices (inputs)
 - Keep the final result in PE accumulators

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

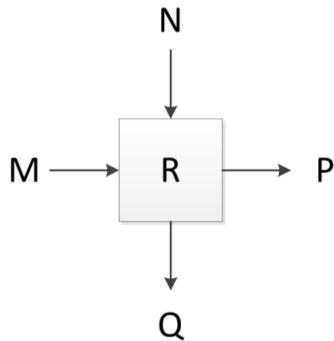
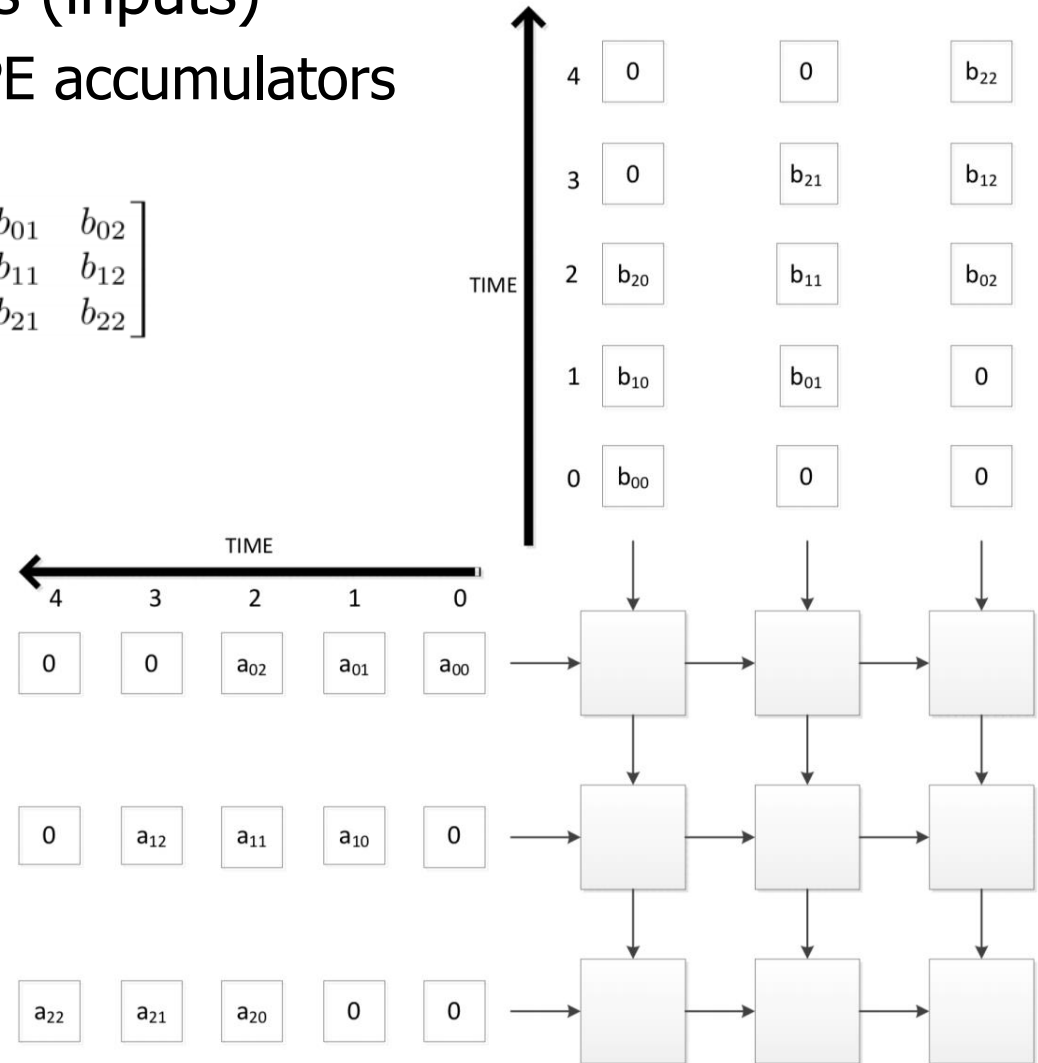


Figure 1: A systolic array processing element

$$P = M$$

$$Q = N$$

$$R = R + M * N$$



An Example Modern Systolic Array: TPU (III)

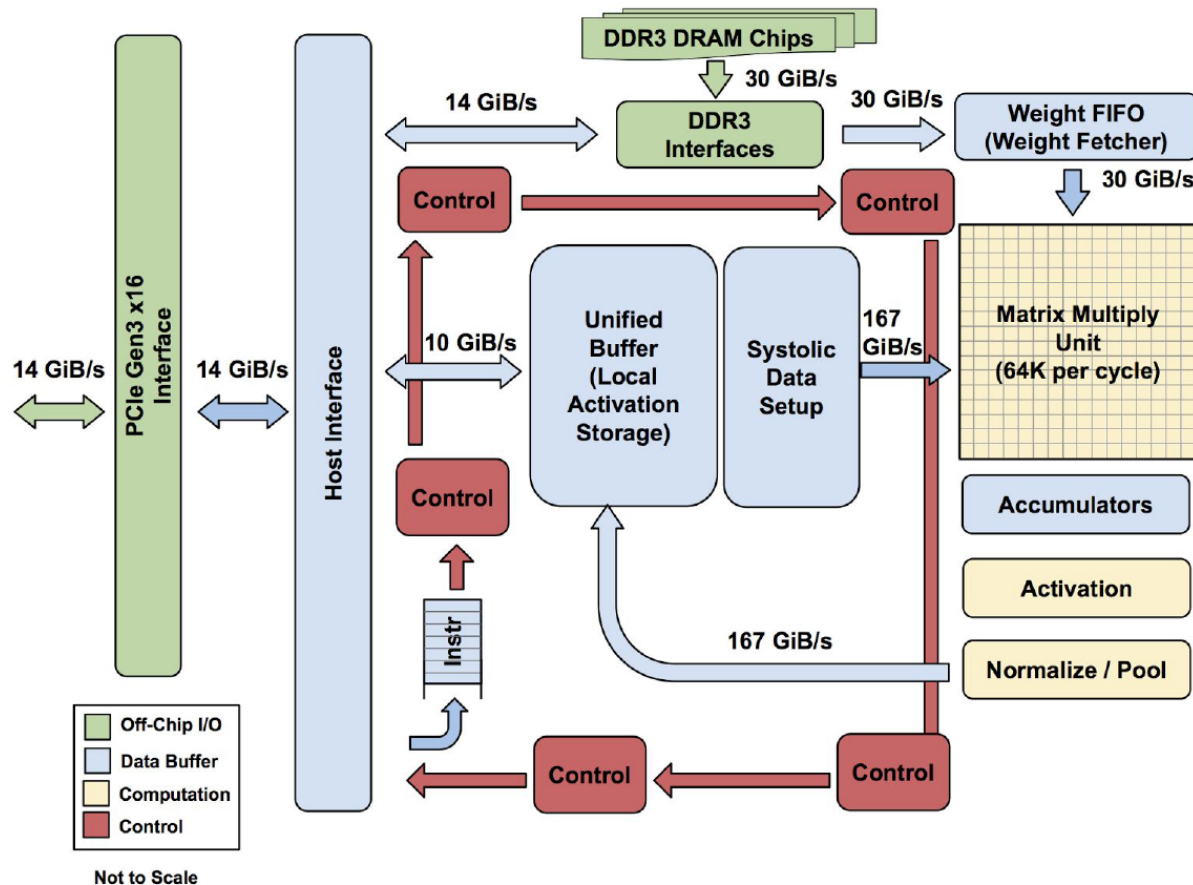


Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.

An Example Modern Systolic Array: TPU2



<https://www.nextplatform.com/2017/05/17/first-depth-look-googles-new-second-generation-tpu/>

4 TPU chips
vs 1 chip in TPU1

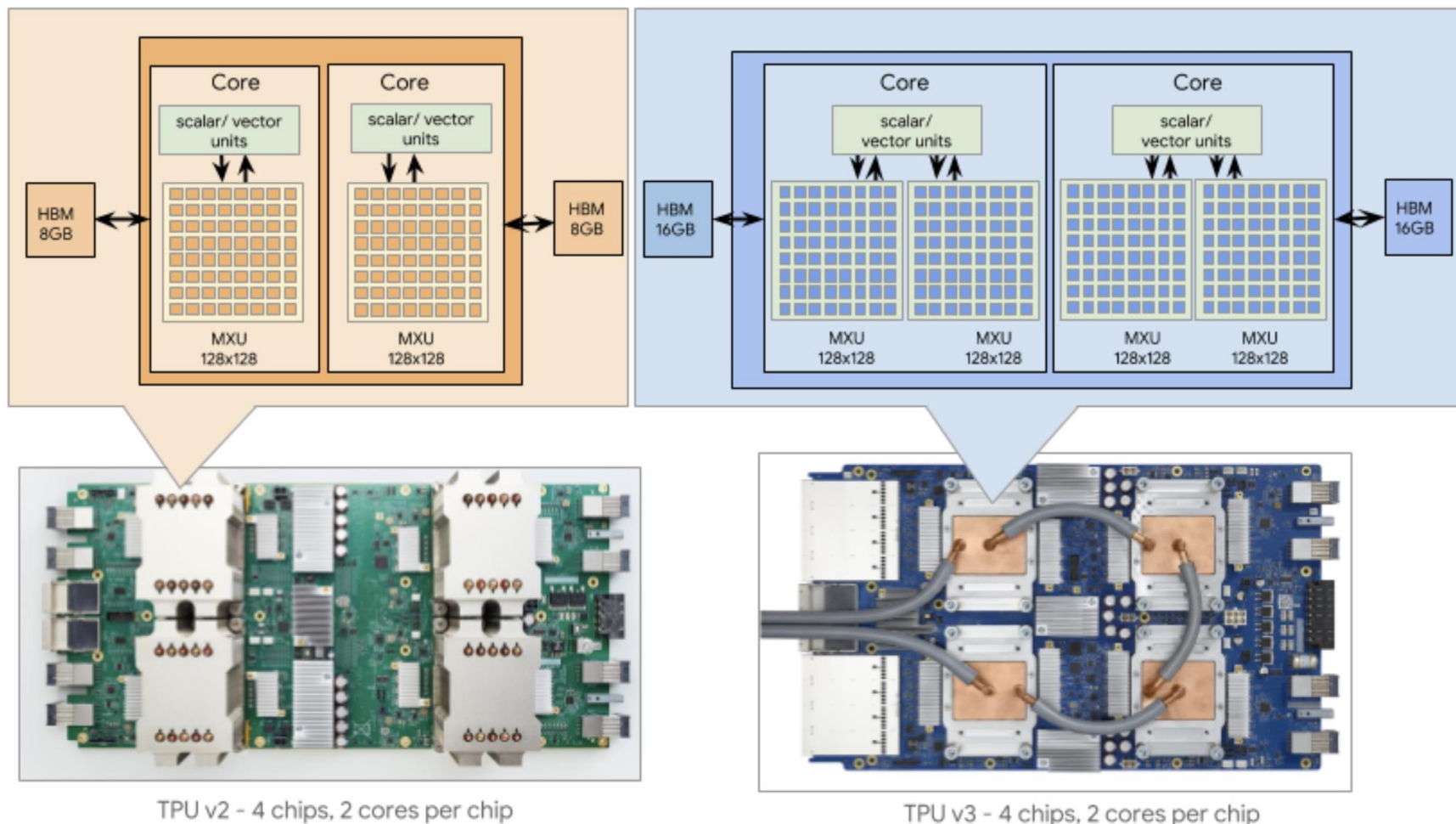
High Bandwidth Memory
vs DDR3

Floating point operations
vs FP16

45 TFLOPS per chip
vs 23 TOPS

Designed for training
and inference
vs only inference

An Example Modern Systolic Array: TPU3

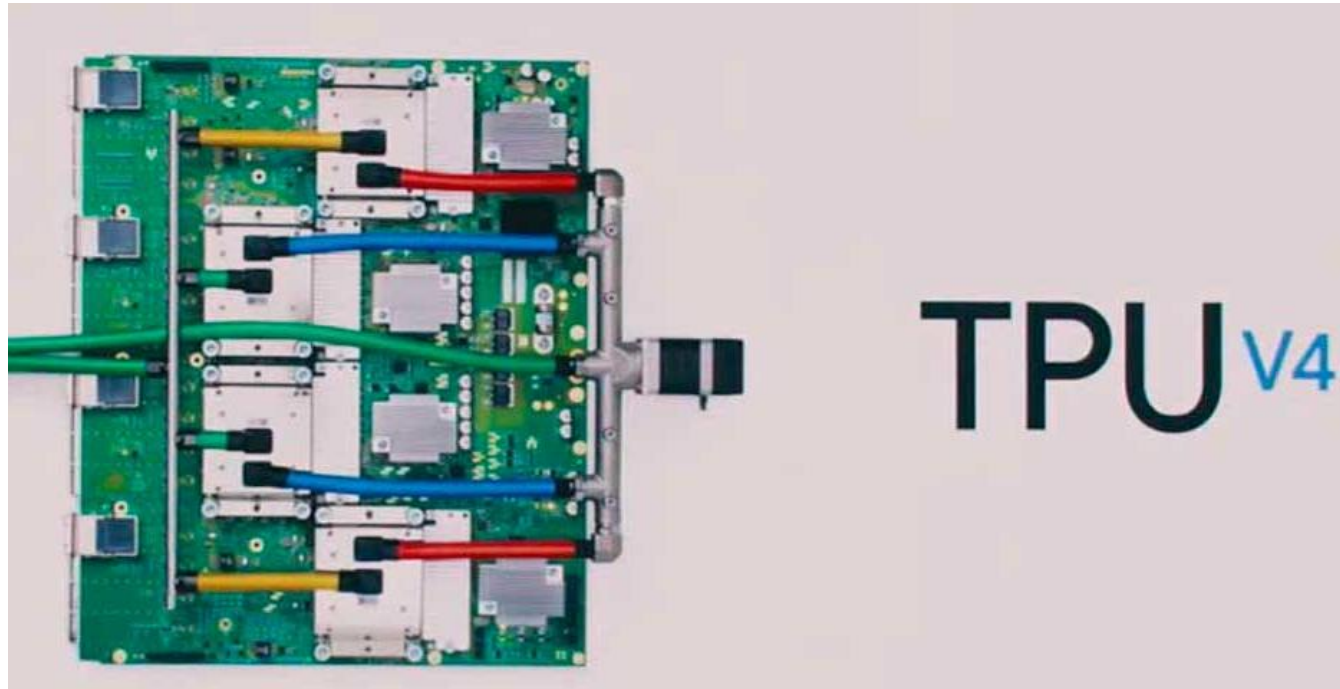


32GB HBM per chip
vs 16GB HBM in TPU2

4 Matrix Units per chip
vs 2 Matrix Units in TPU2

90 TFLOPS per chip
vs 45 TFLOPS in TPU2

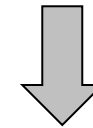
An Example Modern Systolic Array: TPU4



New ML applications (vs. TPU3):

- Computer vision
- Natural Language Processing (NLP)
- Recommender system
- Reinforcement learning that plays Go

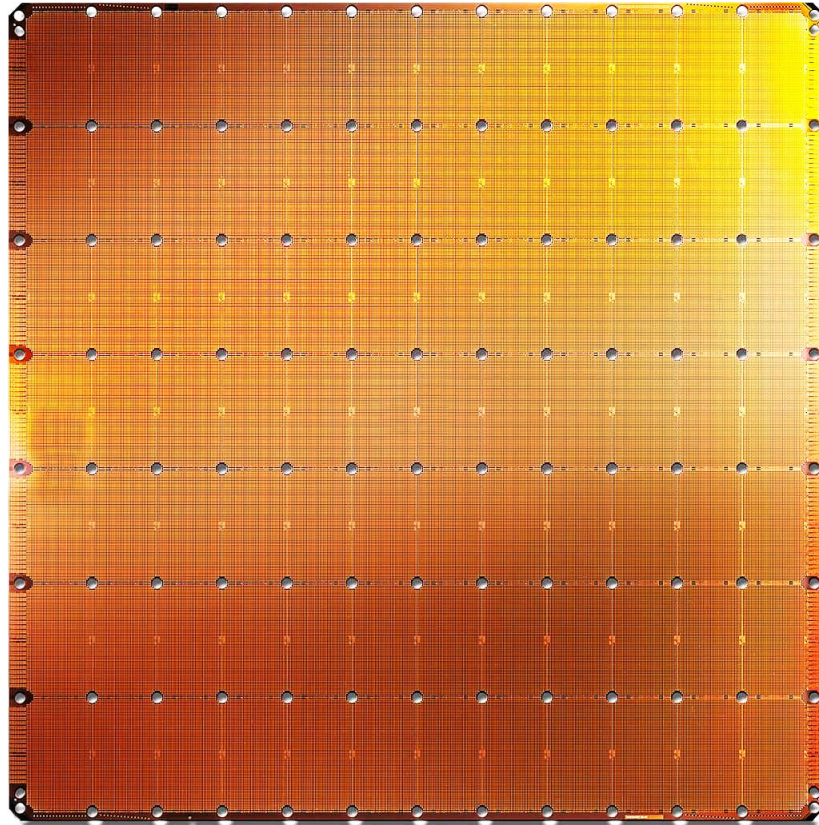
250 TFLOPS per chip in 2021
vs 90 TFLOPS in TPU3



1 ExaFLOPS per board

<https://spectrum.ieee.org/tech-talk/computing/hardware/heres-how-googles-tpu-v4-ai-chip-stacked-up-in-training-tests>

Cerebras's Wafer Scale Engine (2019)



Cerebras WSE

1.2 Trillion transistors
46,225 mm²

- The largest ML accelerator chip
- 400,000 cores



Largest GPU

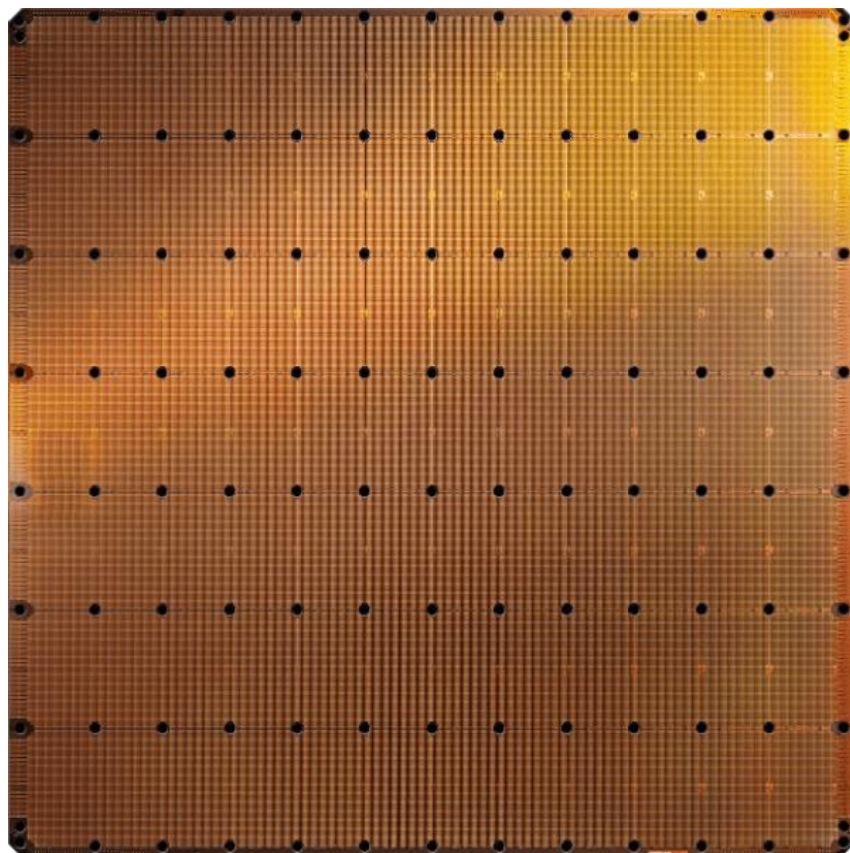
21.1 Billion transistors
815 mm²

NVIDIA TITAN V

<https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning>

<https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning>

Cerebras's Wafer Scale Engine-2 (2021)



Cerebras WSE-2
2.6 Trillion transistors
46,225 mm²

- The largest ML accelerator chip
- 850,000 cores



Largest GPU
54.2 Billion transistors
826 mm²

NVIDIA Ampere GA100

<https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning>

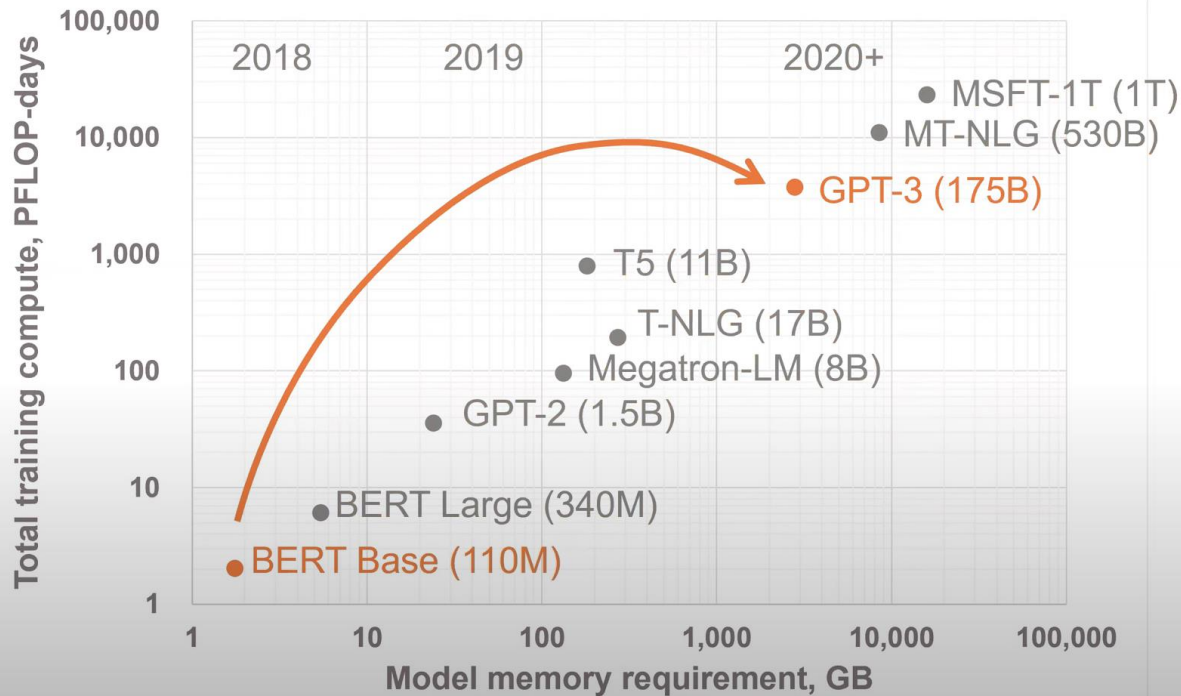
<https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning>

Huge Demand for Performance & Efficiency

Exponential Growth of Neural Networks



Memory and compute requirements



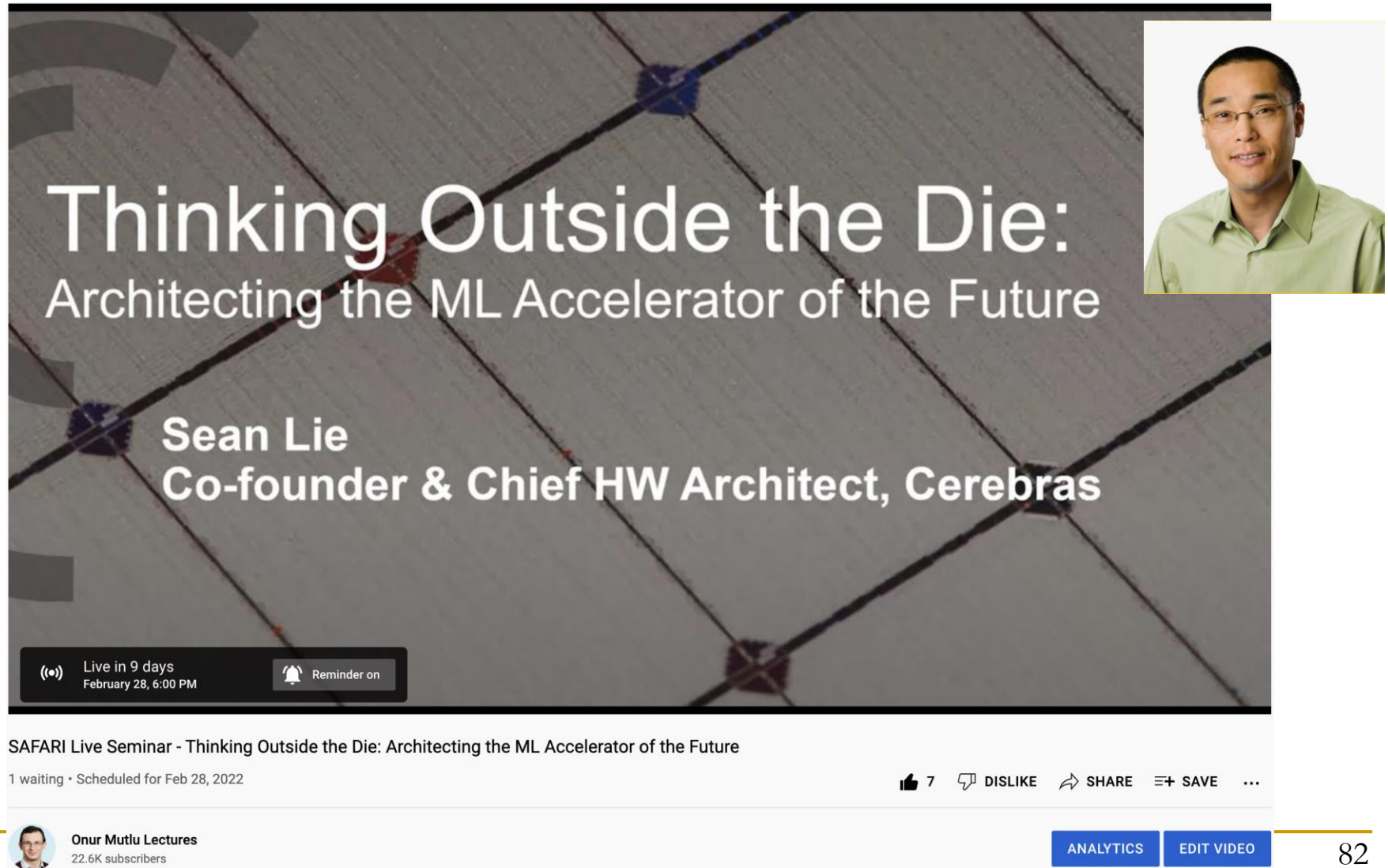
1800x more compute

In just 2 years

Tomorrow, multi-trillion
parameter models

More on the Cerebras WSE

<https://www.youtube.com/watch?v=x2-qB0J7KHw>



The image shows a YouTube video player interface. The main video frame displays a title card with a background of a circuit board. The title is "Thinking Outside the Die: Architecting the ML Accelerator of the Future" in large white text. Below the title, the speaker's name "Sean Lie" and his role "Co-founder & Chief HW Architect, Cerebras" are listed in white text. In the top right corner of the video frame, there is a small inset portrait of Sean Lie, a man with glasses wearing a light green shirt. At the bottom left of the video frame, there is a black bar with white text indicating "Live in 9 days February 28, 6:00 PM" and a bell icon with the text "Reminder on". Below the video frame, the video title "SAFARI Live Seminar - Thinking Outside the Die: Architecting the ML Accelerator of the Future" is displayed. Below the title, it says "1 waiting • Scheduled for Feb 28, 2022". To the right of the title, there are icons for likes (7), dislikes, shares, and a save button. At the bottom left, there is a profile picture of Onur Mutlu Lectures and the text "Onur Mutlu Lectures 22.6K subscribers". At the bottom right, there are two blue buttons labeled "ANALYTICS" and "EDIT VIDEO".

Thinking Outside the Die:
Architecting the ML Accelerator of the Future

Sean Lie
Co-founder & Chief HW Architect, Cerebras

Live in 9 days
February 28, 6:00 PM

Reminder on

SAFARI Live Seminar - Thinking Outside the Die: Architecting the ML Accelerator of the Future

1 waiting • Scheduled for Feb 28, 2022

7 DISLIKE SHARE SAVE ...

Onur Mutlu Lectures
22.6K subscribers

ANALYTICS EDIT VIDEO

Computer Architecture

Lecture 28: VLIW and Systolic Array Architectures

Prof. Onur Mutlu

ETH Zürich

Fall 2022

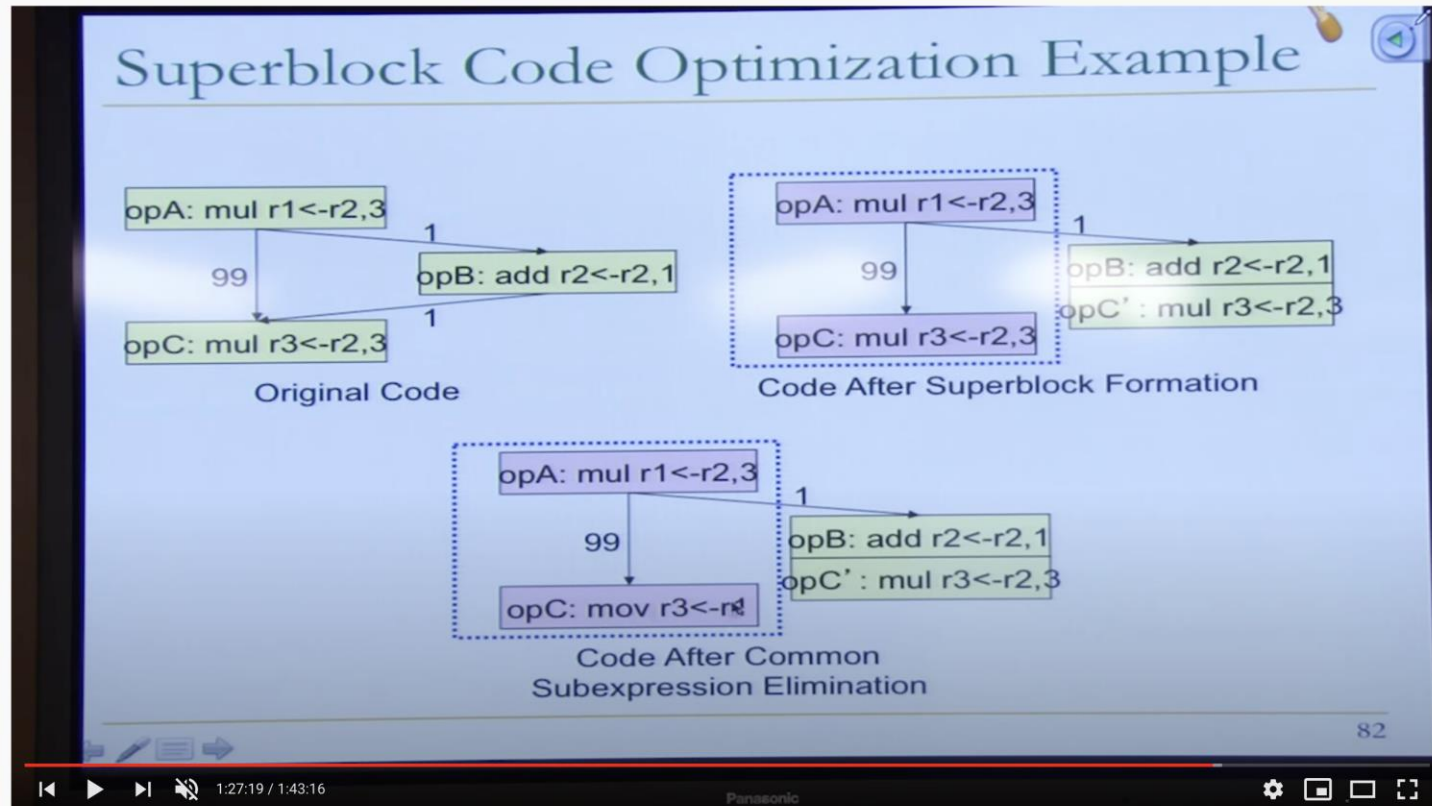
10 January 2023

Backup Slides

(for Further Study)

Issues in Fast & Wide Fetch Engines

These Issues Covered in This Lecture...



18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5

4,696 views • Sep 23, 2015

41 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
23K subscribers

Lecture 5: Advanced Branch Prediction

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 16, 2014.

Lecture 5 slides (pdf): <http://www.ece.cmu.edu/~ece740/f15/li...>

Lecture 5 slides (ppt): <http://www.ece.cmu.edu/~ece740/f15/li...>

<https://www.youtube.com/onurmutlulectures>

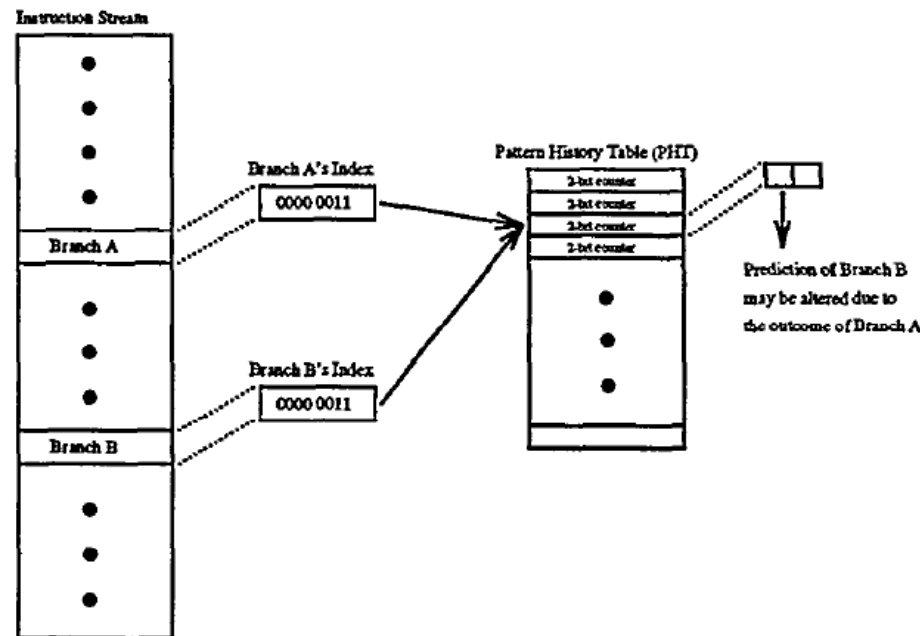
These Issues Covered in This Lecture...

- Computer Architecture, Spring 2015, Lecture 5
 - Advanced Branch Prediction (CMU, Spring 2015)
 - <https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4>

Interference in Branch Predictors

An Issue: Interference in the PHTs

- Sharing the PHTs between histories/branches leads to interference
 - Different branches map to the same PHT entry and modify it
 - Interference can be positive, **negative**, or neutral



- Interference can be eliminated by dedicating a PHT per branch
 - Too much hardware cost
- How else can you eliminate or reduce interference?

Reducing Interference in PHTs (I)

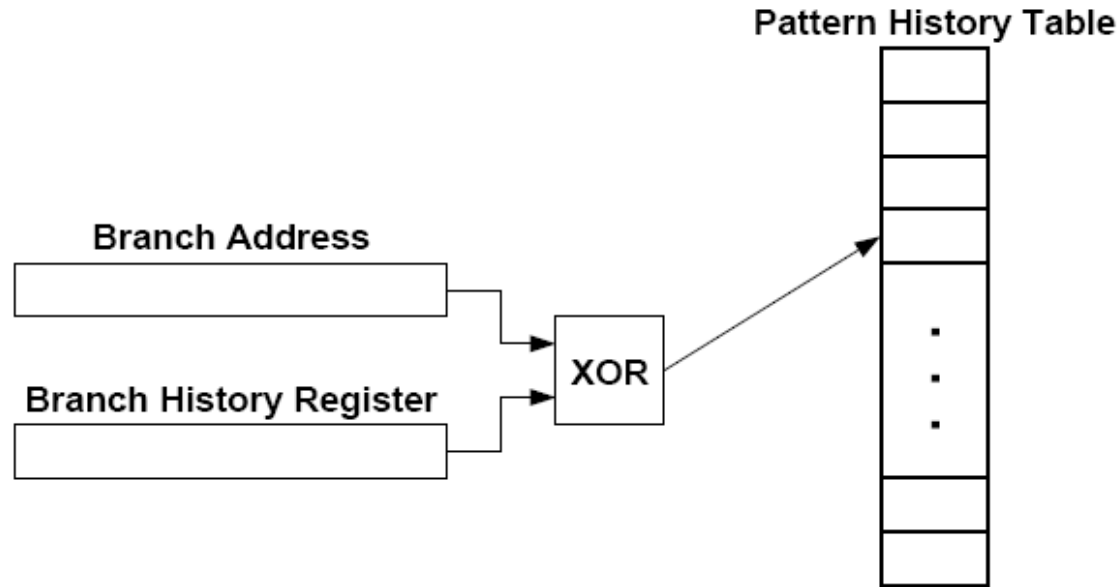
- Increase size of PHT
- Branch filtering
 - Predict highly-biased branches separately so that they do not consume PHT entries
 - E.g., static prediction or BTB based prediction
- Hashing/index-randomization
 - Gshare
 - Gskew
- Agree prediction

Biased Branches and Branch Filtering

- Observation: Many branches are biased in one direction (e.g., 99% taken)
- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing “interference” in branch prediction tables and history registers
- Solution: Detect such biased branches, and predict them with a simpler predictor (e.g., last time, static, ...)
- Chang et al., “Branch classification: a new mechanism for improving branch predictor performance,” MICRO 1994.

Reducing Interference: Gshare

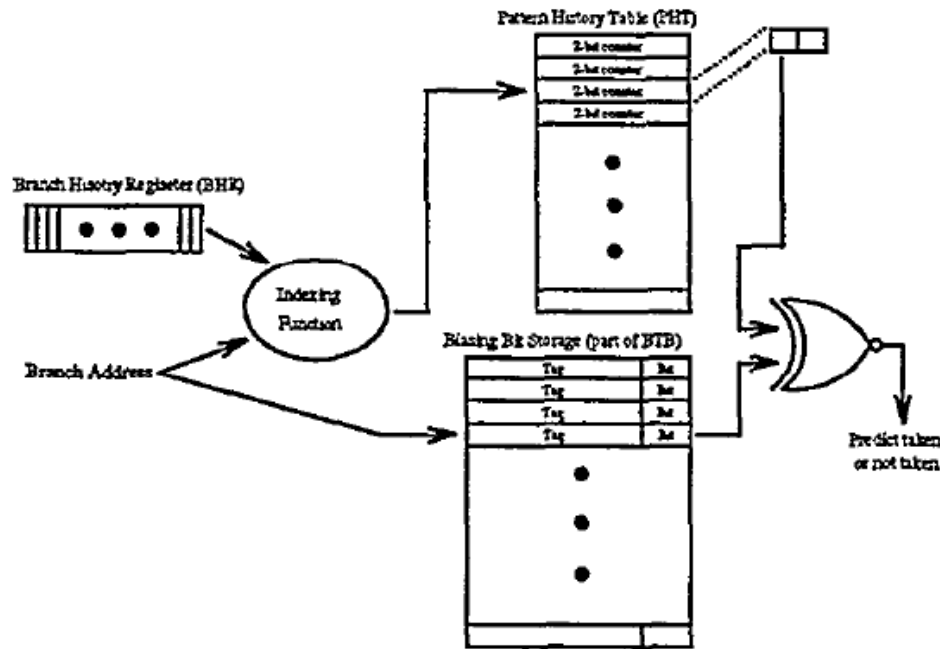
- Idea 1: Randomize the indexing function into the PHT such that probability of two branches mapping to the same entry reduces
 - Gshare predictor: GHR hashed with the Branch PC
 - + Better utilization of PHT + More context information
 - Increases access latency



- McFarling, “Combining Branch Predictors,” DEC WRL Tech Report, 1993.

Reducing Interference: Agree Predictor

- Idea 2: Agree prediction
 - Each branch has a “bias” bit associated with it in BTB
 - Ideally, most likely outcome for the branch
 - High bit of the PHT counter indicates whether or not the prediction agrees with the bias bit (not whether or not prediction is taken)
- + Reduces negative interference (Why???)
- Requires determining bias bits (compiler vs. hardware)



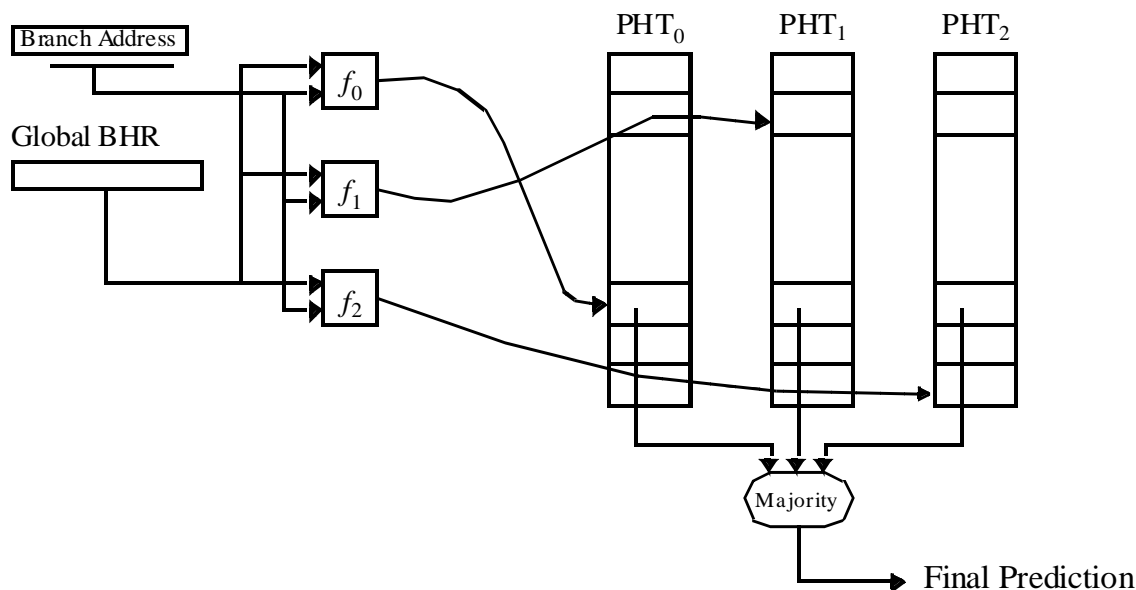
Sprangle et al., “The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference,” ISCA 1997.

Why Does Agree Prediction Make Sense?

- Assume two branches have taken rates of 85% and 15%.
- Assume they conflict in the PHT
- Let's compute the **probability they have opposite outcomes**
 - Baseline predictor:
 - $P(b1 \text{ T}, b2 \text{ NT}) + P(b1 \text{ NT}, b2 \text{ T})$
 $= (85\% * 85\%) + (15\% * 15\%) = 74.5\%$
 - Agree predictor:
 - Assume bias bits are set to T (b1) and NT (b2)
 - $P(b1 \text{ agree}, b2 \text{ disagree}) + P(b1 \text{ disagree}, b2 \text{ agree})$
 $= (85\% * 15\%) + (15\% * 85\%) = 25.5\%$
- Works because most branches are biased (not 50% taken)

Reducing Interference: Gskew

- Idea 3: Gskew predictor
 - Multiple PHTs
 - Each indexed with a different type of hash function
 - Final prediction is a majority vote
- + Distributes interference patterns in a more randomized way (interfering patterns less likely in different PHTs at the same time)
- More complexity (due to multiple PHTs, hash functions)



Seznec, “An optimized 2bcgskew branch predictor,” IRISA Tech Report 1993.

Michaud, “Trading conflict and capacity aliasing in conditional branch predictors,” ISCA 1997

More Techniques to Reduce PHT Interference

- The bi-mode predictor
 - Separate PHTs for mostly-taken and mostly-not-taken branches
 - Reduces negative aliasing between them
 - Lee et al., “The bi-mode branch predictor,” MICRO 1997.
- The YAGS predictor
 - Use a small tagged “cache” to predict branches that have experienced interference
 - Aims to not to mispredict them again
 - Eden and Mudge, “The YAGS branch prediction scheme,” MICRO 1998.
- Alpha EV8 (21464) branch predictor
 - Seznec et al., “Design tradeoffs for the Alpha EV8 conditional branch predictor,” ISCA 2002.

Another Direction: Helper Threading

- Idea: Pre-compute the outcome of the branch with a separate, customized thread (i.e., a helper thread)

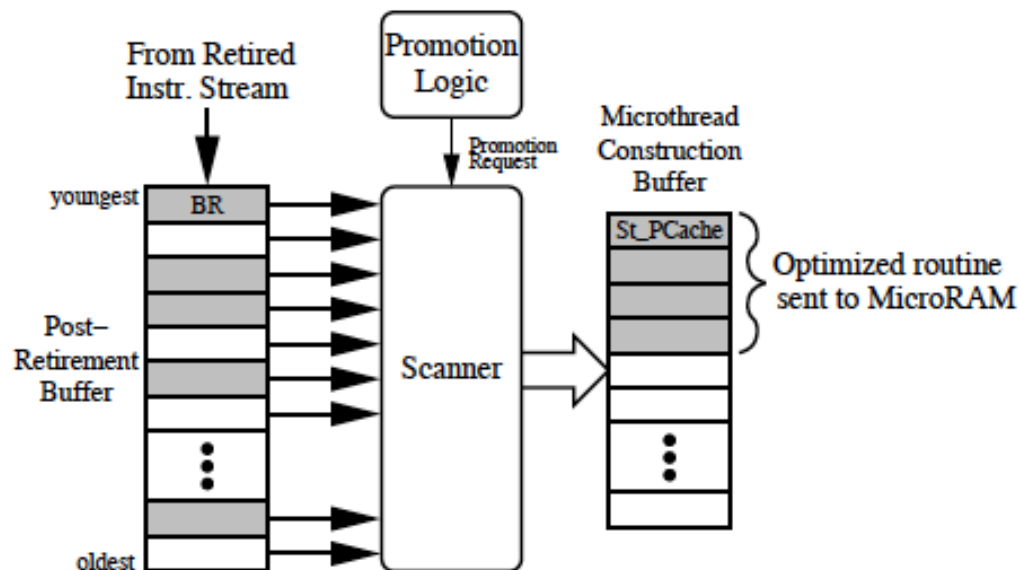


Figure 3. The Microthread Builder

- Chappell et al., “**Difficult-Path Branch Prediction Using Subordinate Microthreads**,” ISCA 2002.
- Chappell et al., “**Simultaneous Subordinate Microthreading**,” ISCA 1999.

Issues in Wide & Fast Fetch

I-Cache Line and Way Prediction

- Problem: Complex branch prediction can take too long (many cycles)
- Goal
 - ❑ Quickly generate (a reasonably accurate) next fetch address
 - ❑ Enable the fetch engine to run at high frequencies
 - ❑ Override the quick prediction with more sophisticated prediction
- Idea: Get the predicted next cache line and way at the time you fetch the current cache line
- Example Mechanism (e.g., Alpha 21264)
 - ❑ Each cache line tells which line/way to fetch next (prediction)
 - ❑ On a fill, line/way predictor points to next sequential line
 - ❑ On branch resolution, line/way predictor is updated
 - ❑ If line/way prediction is incorrect, one cycle is wasted

Alpha 21264 Line & Way Prediction

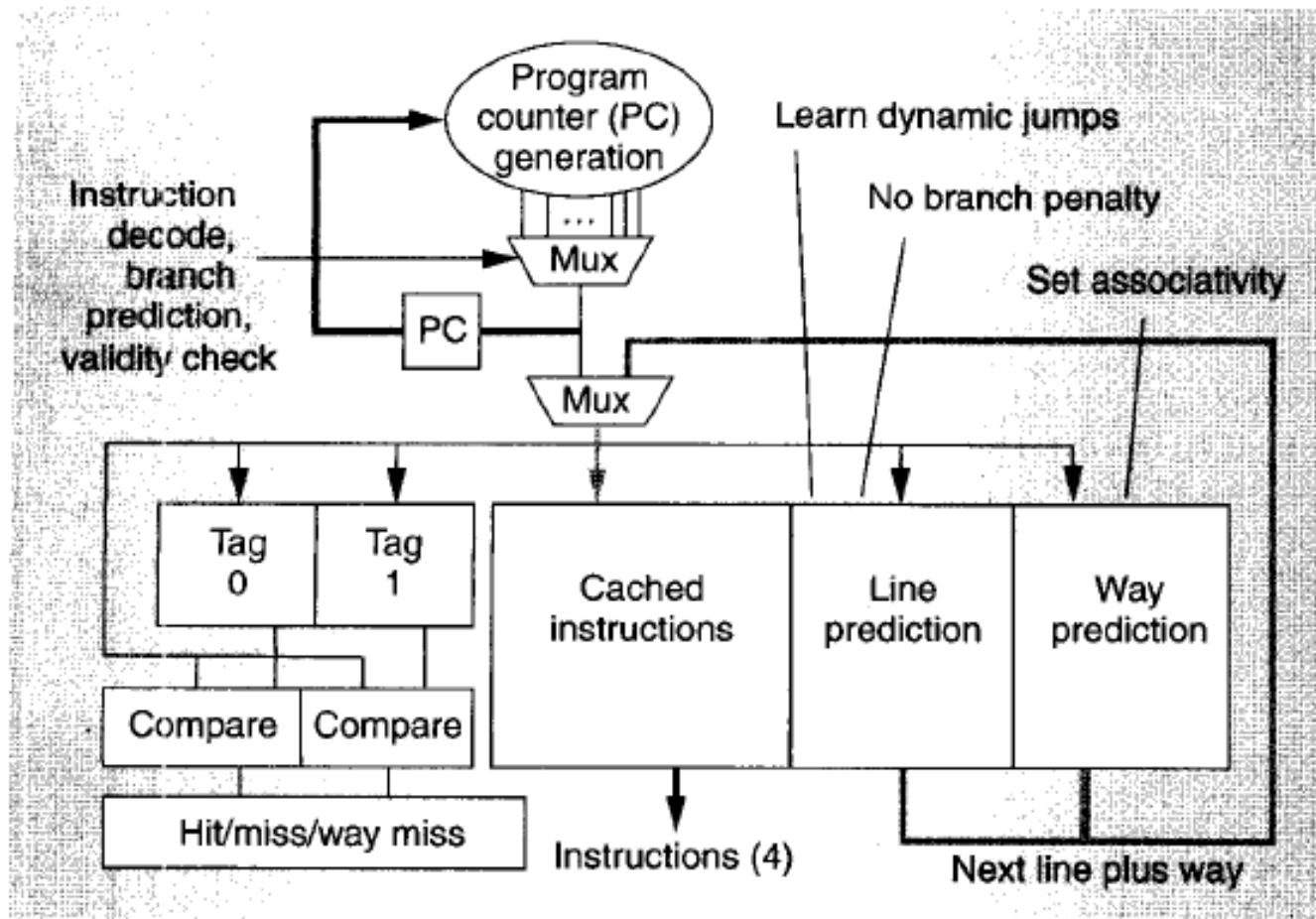
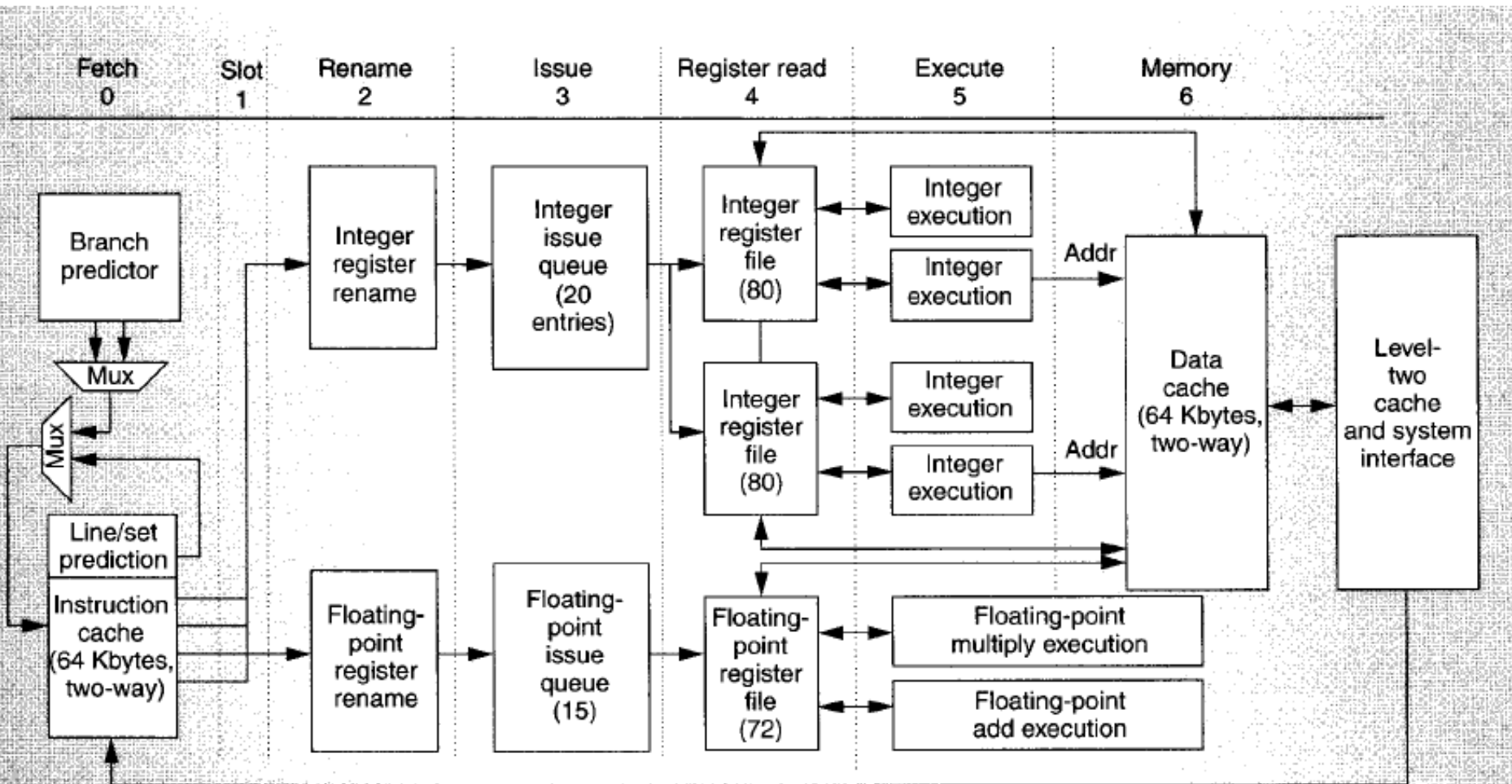


Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.

Alpha 21264 Line & Way Prediction



Issues in Wide Fetch Engines

- Wide Fetch: Fetch multiple instructions per cycle
- Superscalar
- VLIW
- SIMT (GPUs' single-instruction multiple thread model)
- Wide fetch engines suffer from the branch problem:
 - How do you feed the wide pipeline with useful instructions in a single cycle?
 - What if there is a taken branch in the "fetch packet"?
 - What if there are "multiple (taken) branches" in the "fetch packet"?

Fetching Multiple Instructions Per Cycle

- Two problems

1. **Alignment** of instructions in I-cache

- ❑ What if there are not enough (N) instructions in the cache line to supply the fetch width?

2. **Fetch break**: Branches present in the fetch block

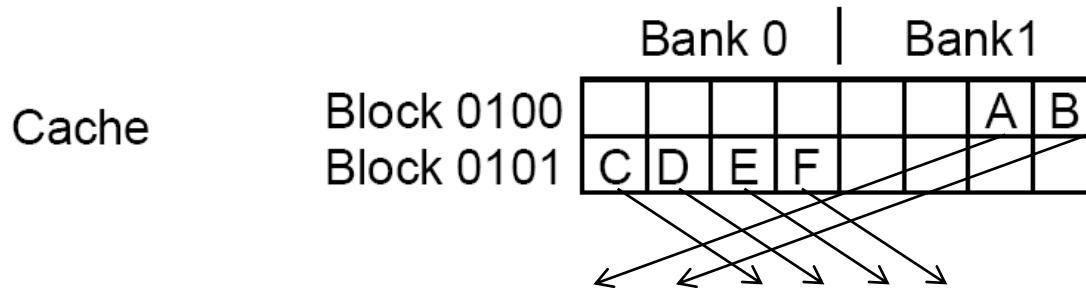
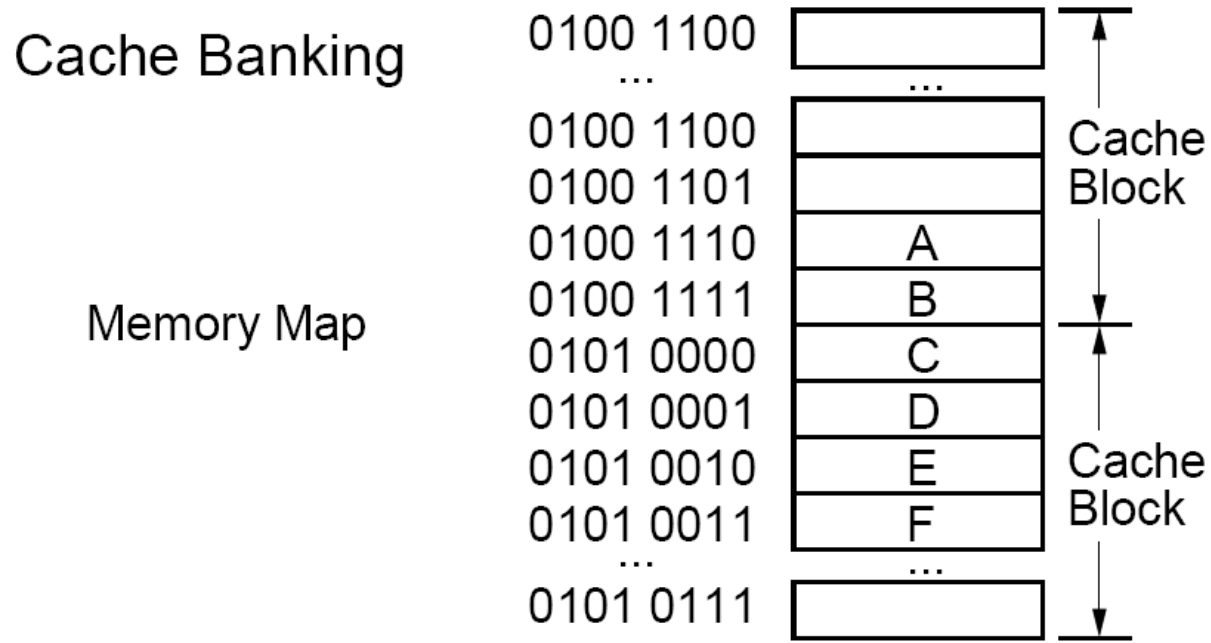
- ❑ Fetching sequential instructions in a single cycle is easy
- ❑ What if there is a control flow instruction in the N instructions?
- ❑ Problem: **The direction of the branch is not known but we need to fetch more instructions**

- These can cause effective fetch width < peak fetch width

Wide Fetch Solutions: Alignment

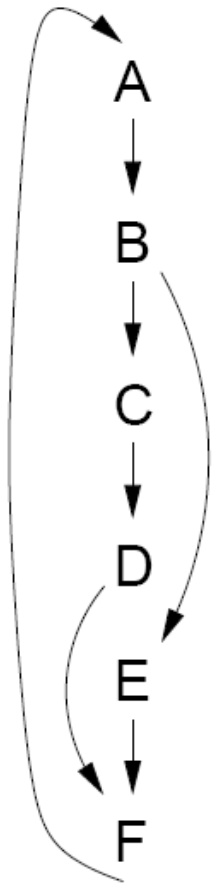
- **Large cache blocks:** Hope N instructions contained in the block
- **Split-line fetch:** If address falls into second half of the cache block, fetch the first half of next cache block as well
 - ❑ Enabled by banking of the cache
 - ❑ Allows sequential fetch across cache blocks in one cycle
 - ❑ Intel Pentium and AMD K5

Split Line Fetch



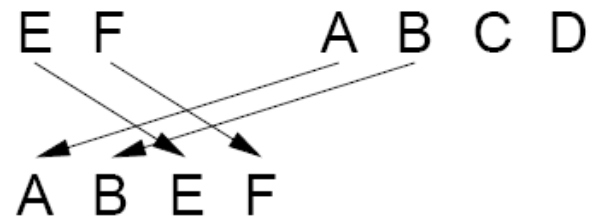
Need alignment logic:

Short Distance Predicted-Taken Branches

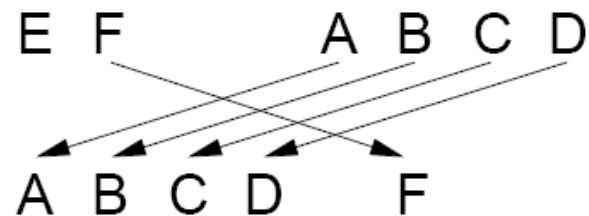


	Bank 0				Bank 1			
Block 0100					A	B	C	D
Block 0101	E	F						

First Iteration (Branch B taken to E)



Second Iteration (Branch B fall through to C)



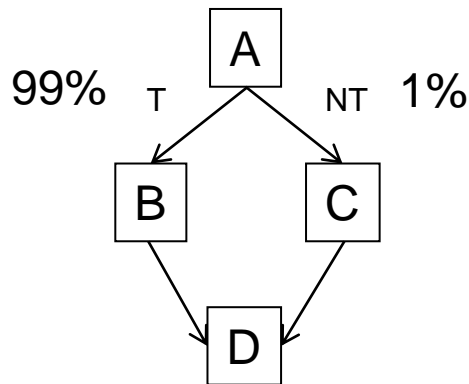
Techniques to Reduce Fetch Breaks

- Compiler
 - Code reordering (basic block reordering)
 - Superblock
- Hardware
 - Trace cache
- Hardware/software cooperative
 - Block structured ISA

Basic Block Reordering

- Not-taken control flow instructions not a problem: no fetch break: **make the likely path the not-taken path**
- Idea: **Convert taken branches to not-taken ones**
 - i.e., **reorder basic blocks** (after profiling)
 - Basic block: code with a single entry and single exit point

Control Flow Graph



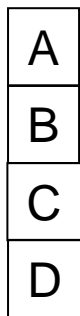
Code Layout 1



Code Layout 2



Code Layout 3



- Code Layout 1 leads to the fewest fetch breaks

Basic Block Reordering

- Pettis and Hansen, “**Profile Guided Code Positioning**,” PLDI 1990.
- Advantages:
 - + Reduced fetch breaks (assuming profile behavior matches runtime behavior of branches)
 - + Increased I-cache hit rate
 - + Reduced page faults
- Disadvantages:
 - Dependent on compile-time profiling
 - Does not help if branches are not biased
 - Requires recompilation

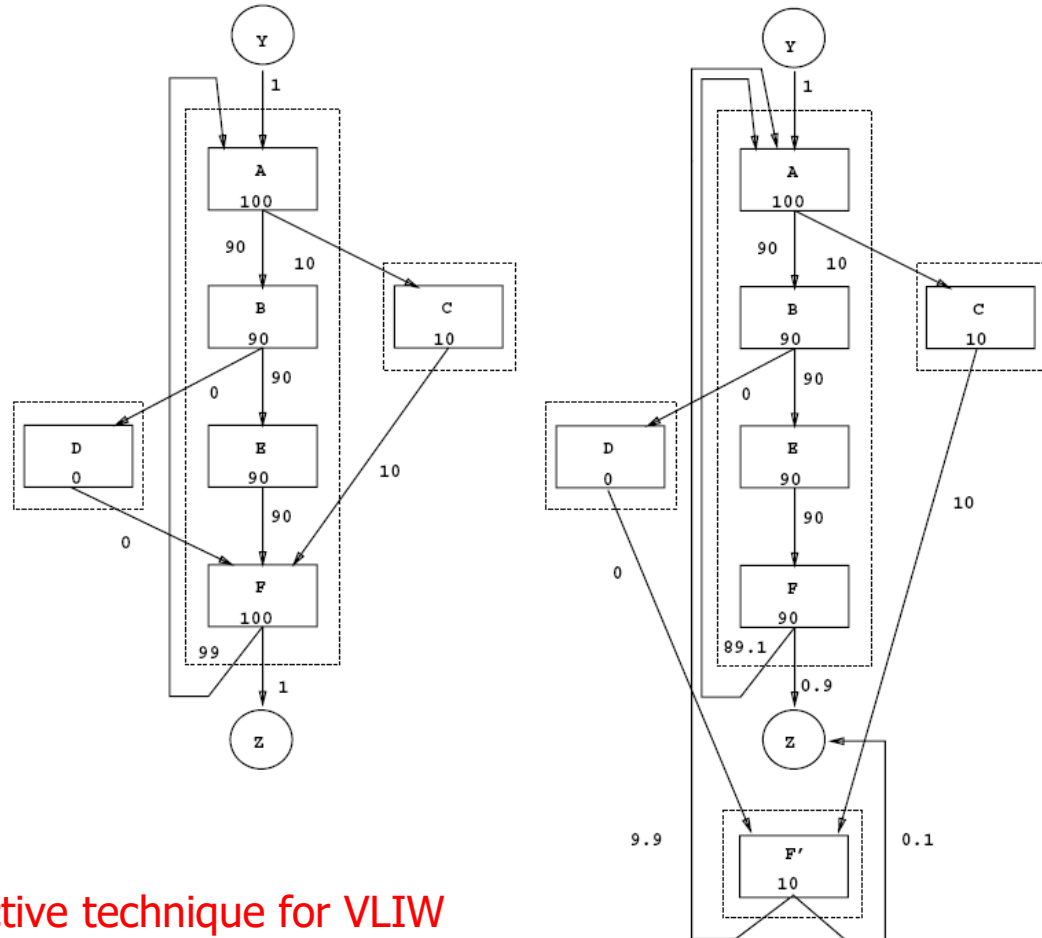
Superblock

- Idea: Combine frequently executed basic blocks such that they form a **single-entry multiple exit larger block**, which is likely executed as straight-line code

- + Helps wide fetch
- + Enables aggressive compiler optimizations and code reordering within the superblock

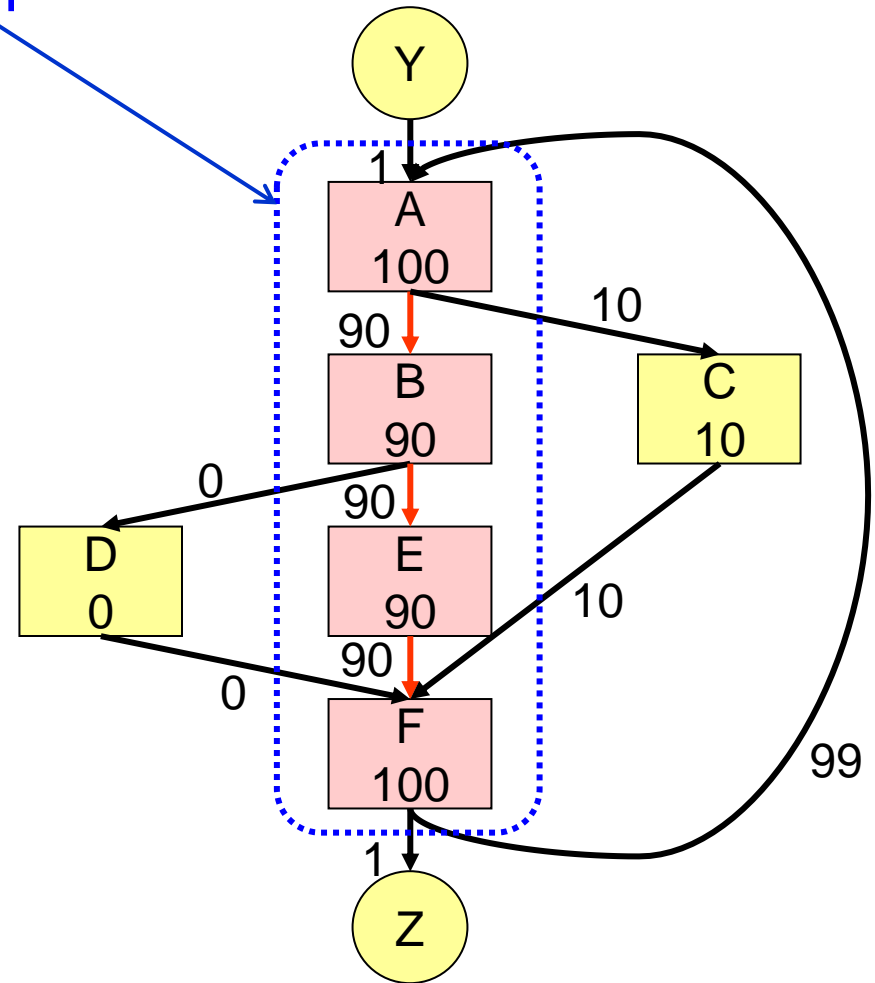
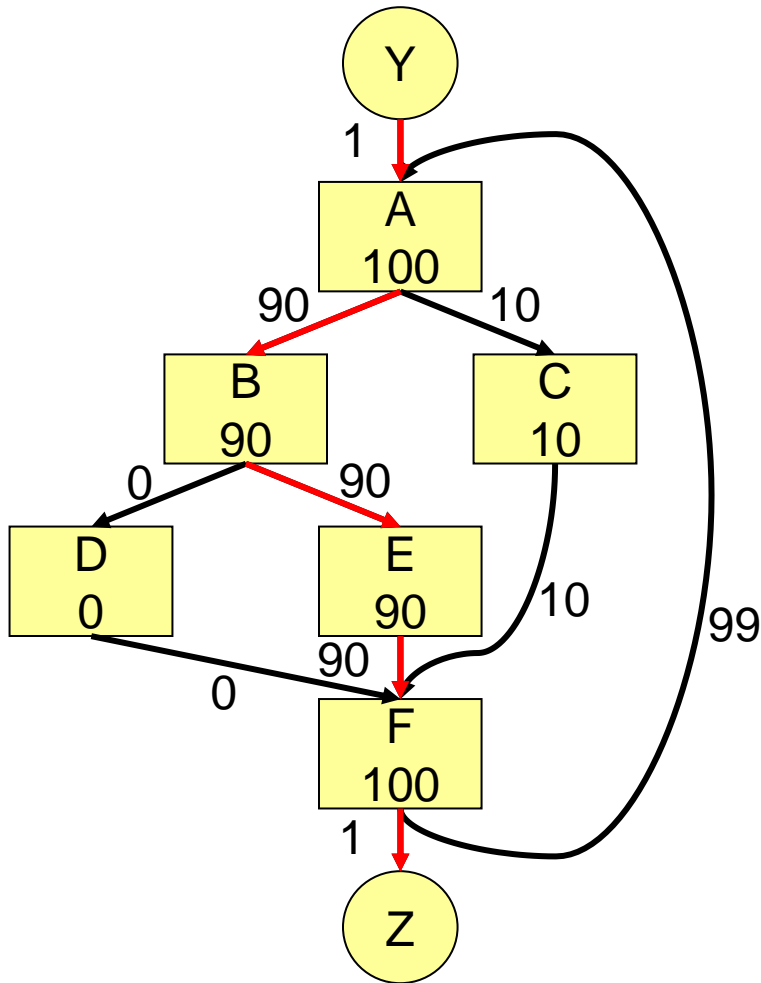
- Increased code size
- Profile dependent
- Requires recompilation

- Hwu et al. “**The Superblock: An effective technique for VLIW and superscalar compilation**,” Journal of Supercomputing, 1993.

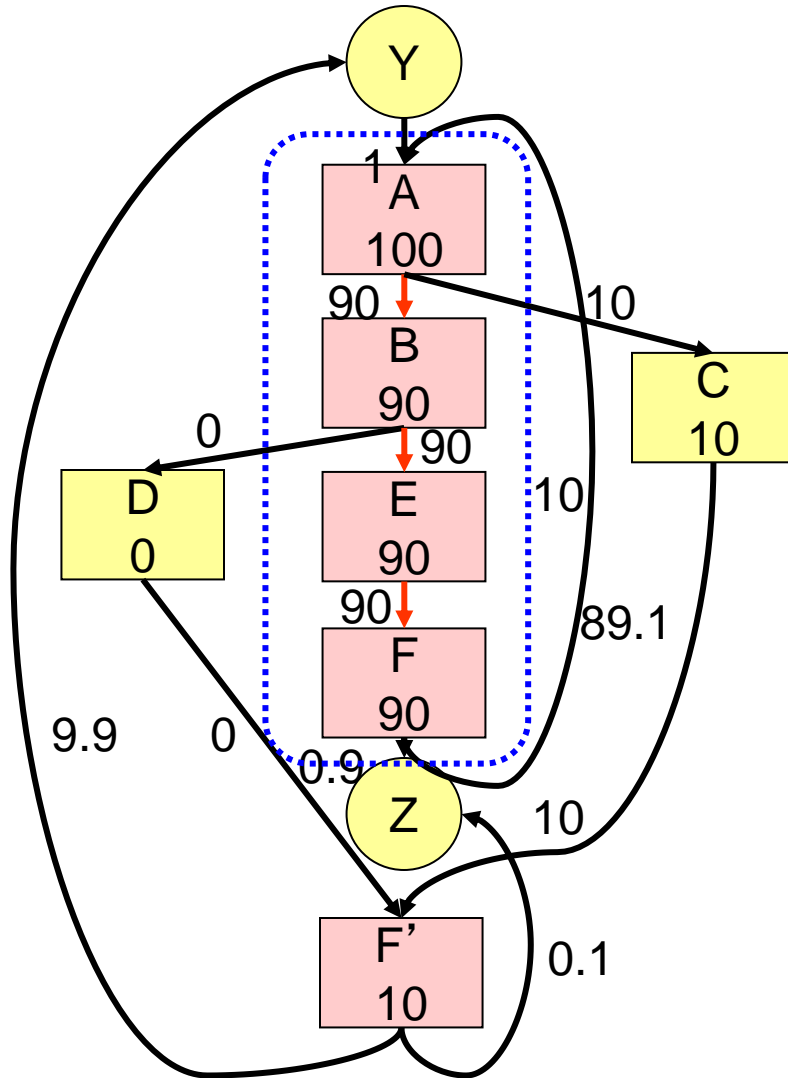


Superblock Formation (I)

Is this a superblock?



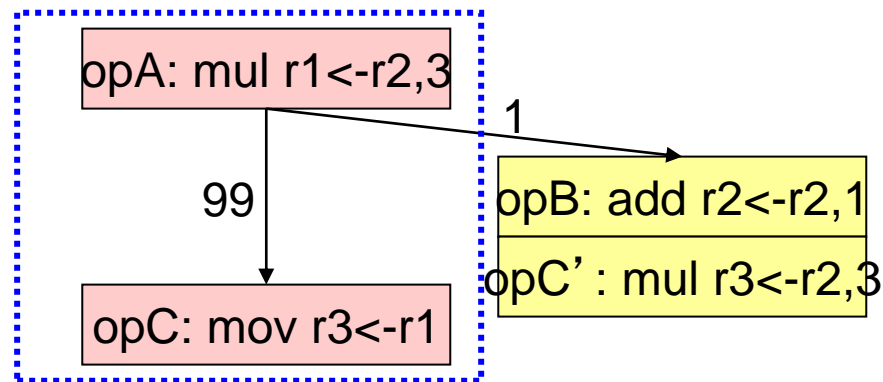
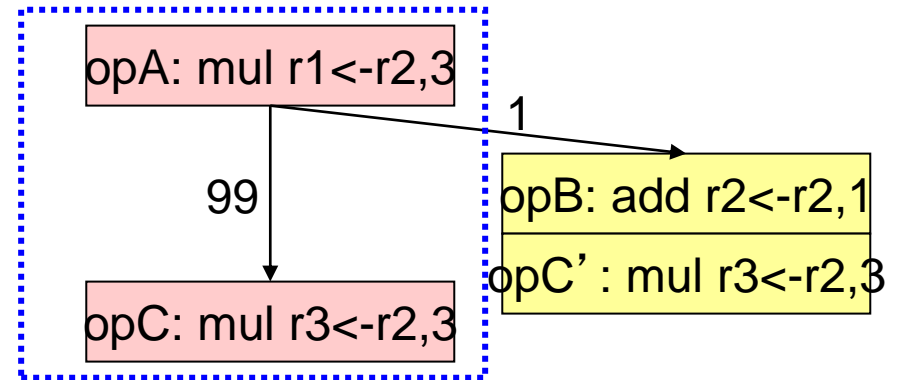
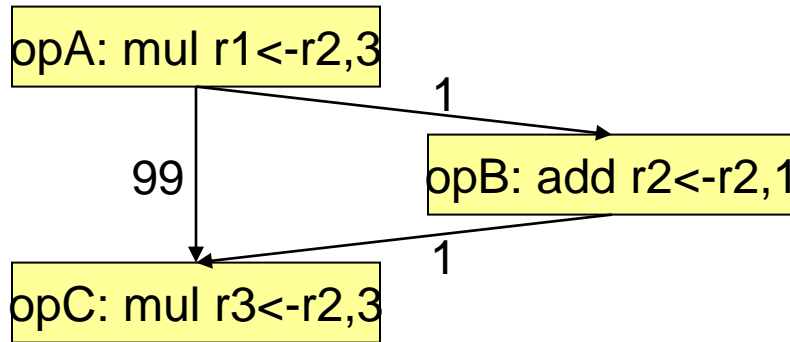
Superblock Formation (II)



Tail duplication:

duplication of basic blocks
after a side entrance to
eliminate side entrances
→ transforms
a **trace** into a **superblock**.

Superblock Code Optimization Example

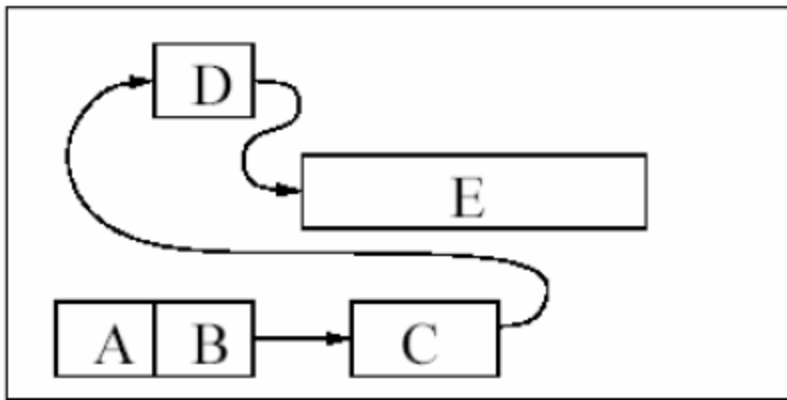


Techniques to Reduce Fetch Breaks

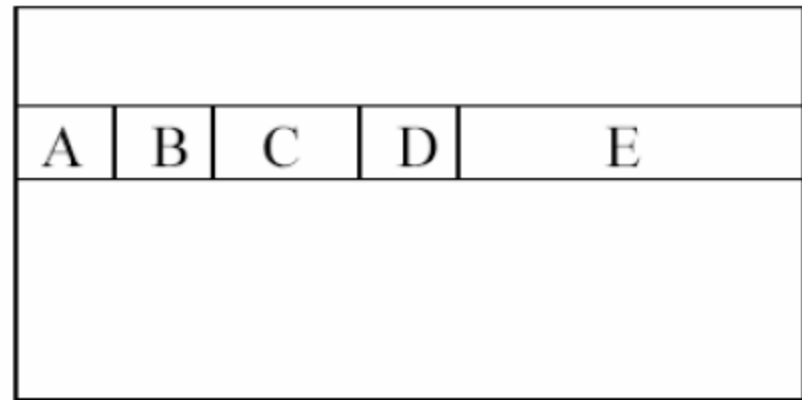
- Compiler
 - Code reordering (basic block reordering)
 - Superblock
- Hardware
 - Trace cache
- Hardware/software cooperative
 - Block structured ISA

Trace Cache: Basic Idea

- A trace is a sequence of executed instructions.
- It is specified by a start address and the branch outcomes of control transfer instructions.
- Traces repeat: programs have frequently executed paths
- Trace cache idea: Store the dynamic instruction sequence in the same physical location.



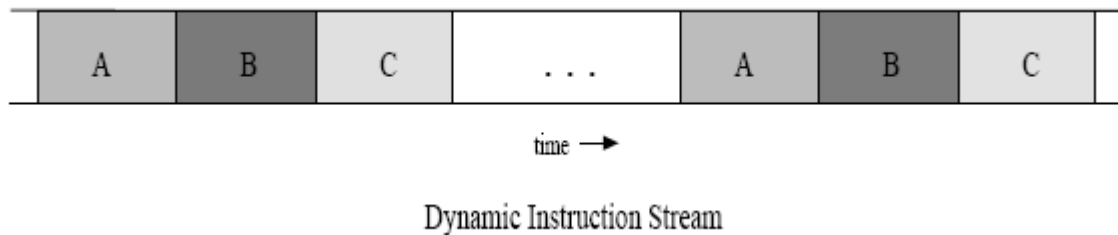
(a) Instruction cache.



(b) Trace cache.

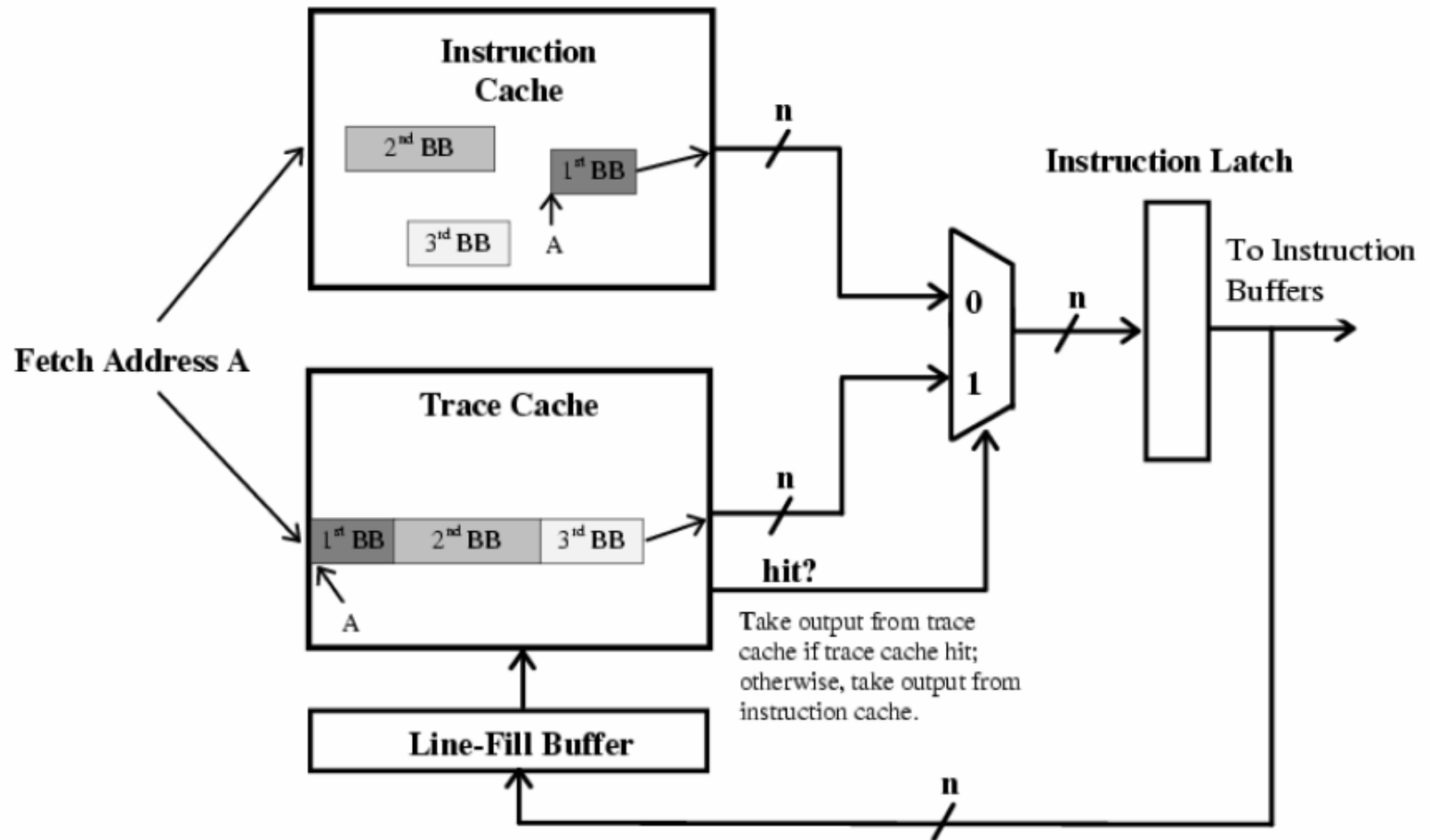
Reducing Fetch Breaks: Trace Cache

- Dynamically determine the basic blocks that are executed consecutively
- Trace: Consecutively executed basic blocks
- Idea: Store consecutively-executed basic blocks in physically-contiguous internal storage (called trace cache)

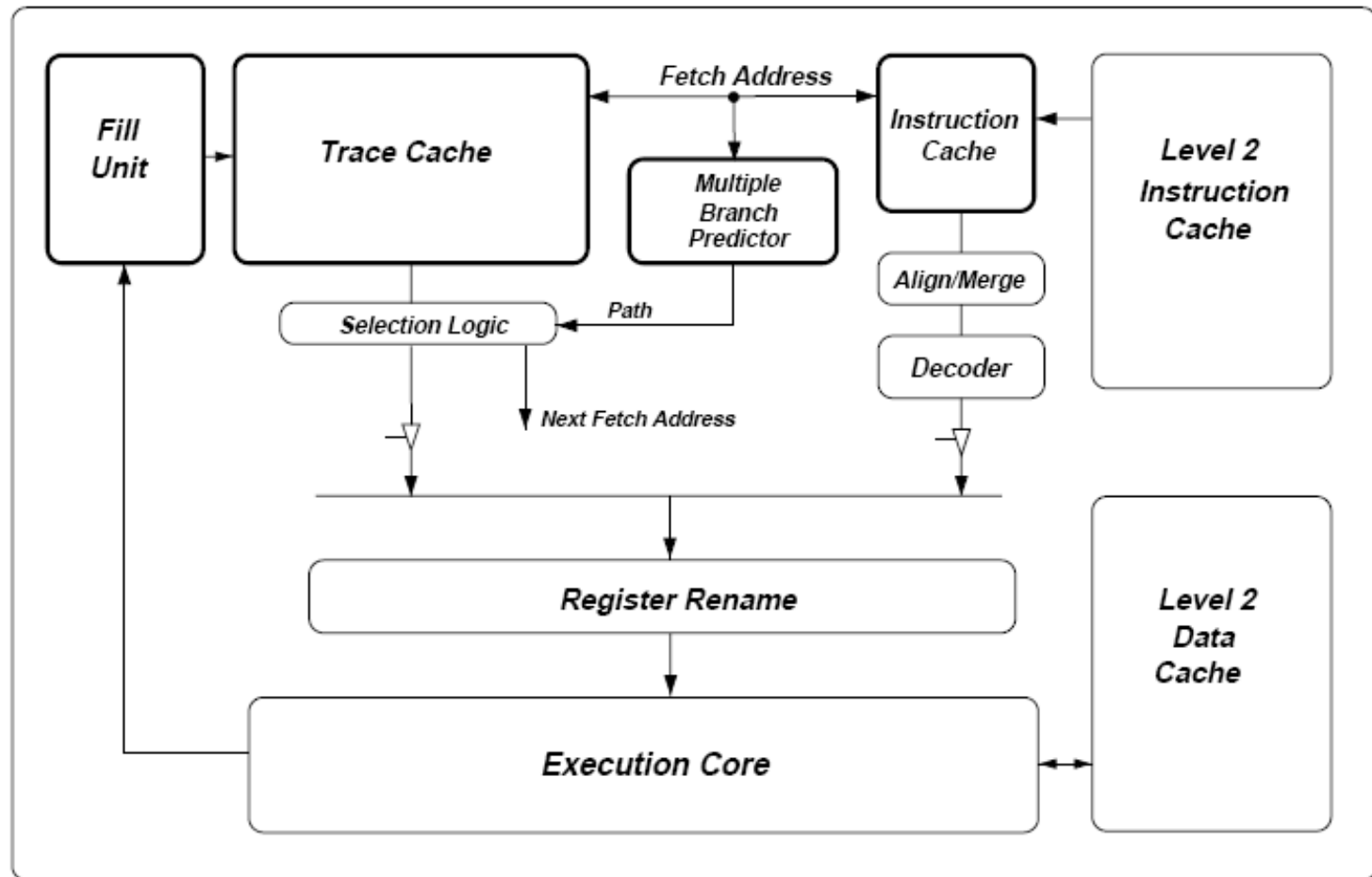


- Basic trace cache operation:
 - Fetch from consecutively-stored basic blocks (predict next trace or branches)
 - Verify the executed branch directions with the stored ones
 - If mismatch, flush the remaining portion of the trace
- Rotenberg et al., “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching,” MICRO 1996.
- Patel et al., “Critical Issues Regarding the Trace Cache Fetch Mechanism,” Umich TR, 1997.

Trace Cache: Example



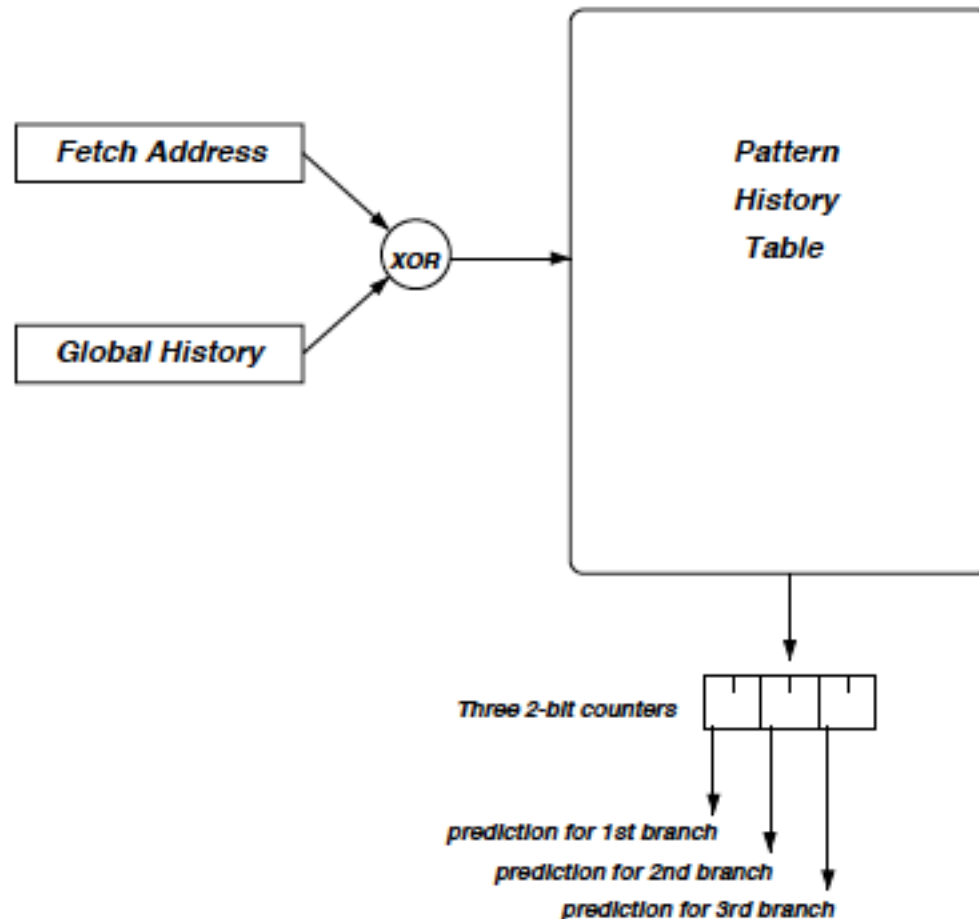
An Example Trace Cache Based Processor



- From Patel's PhD Thesis: "**Trace Cache Design for Wide Issue Superscalar Processors**," University of Michigan, 1999.

Multiple Branch Predictor

- S. Patel, “Trace Cache Design for Wide Issue Superscalar Processors,” PhD Thesis, University of Michigan, 1999.

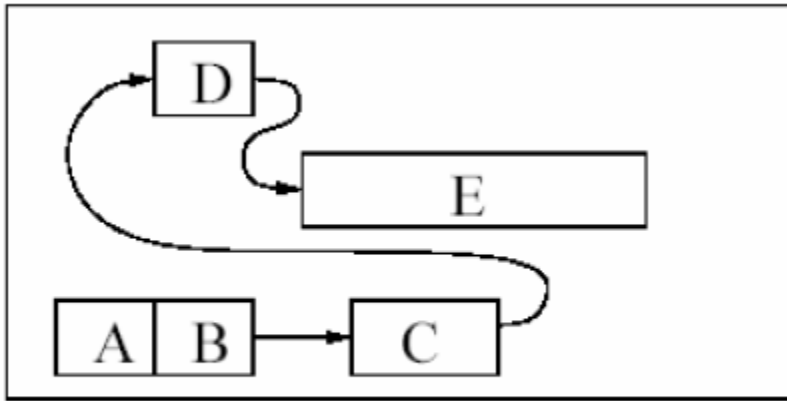


What Does A Trace Cache Line Store?

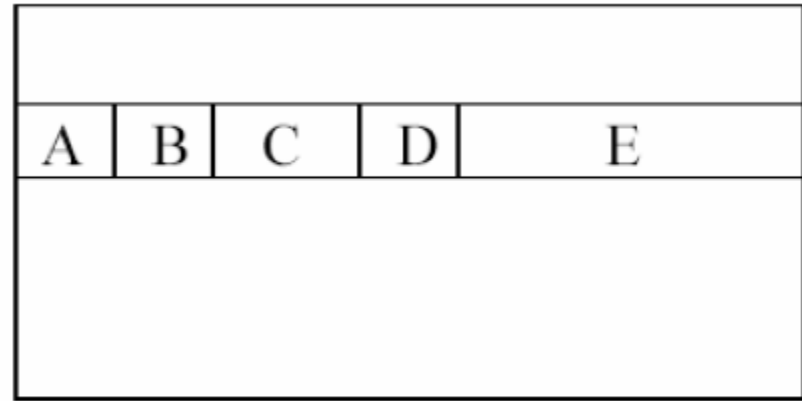
- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.
- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.
- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack provides the next fetch address.

- Patel et al., “Critical Issues Regarding the Trace Cache Fetch Mechanism,” Umich TR, 1997.

Trace Cache: Advantages/Disadvantages



(a) Instruction cache.



(b) Trace cache.

- + Reduces fetch breaks (assuming branches are biased)
- + No need for decoding (instructions can be stored in decoded form)
- + Can enable dynamic optimizations within a trace
- Requires hardware to form traces (more complexity) → called fill unit
- Results in duplication of the same basic blocks in the cache
- Can require the prediction of multiple branches per cycle
 - If multiple cached traces have the same start address
 - What if XYZ and XYT are both likely traces?

Intel Pentium 4 Trace Cache

- A 12K-uop trace cache replaces the L1 I-cache
- Trace cache stores decoded and cracked instructions
 - Micro-operations (uops): returns 6 uops every other cycle
- x86 decoder can be simpler and slower
- A. Peleg, U. Weiser; "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", United States Patent No. 5,381,533, Jan 10, 1995

